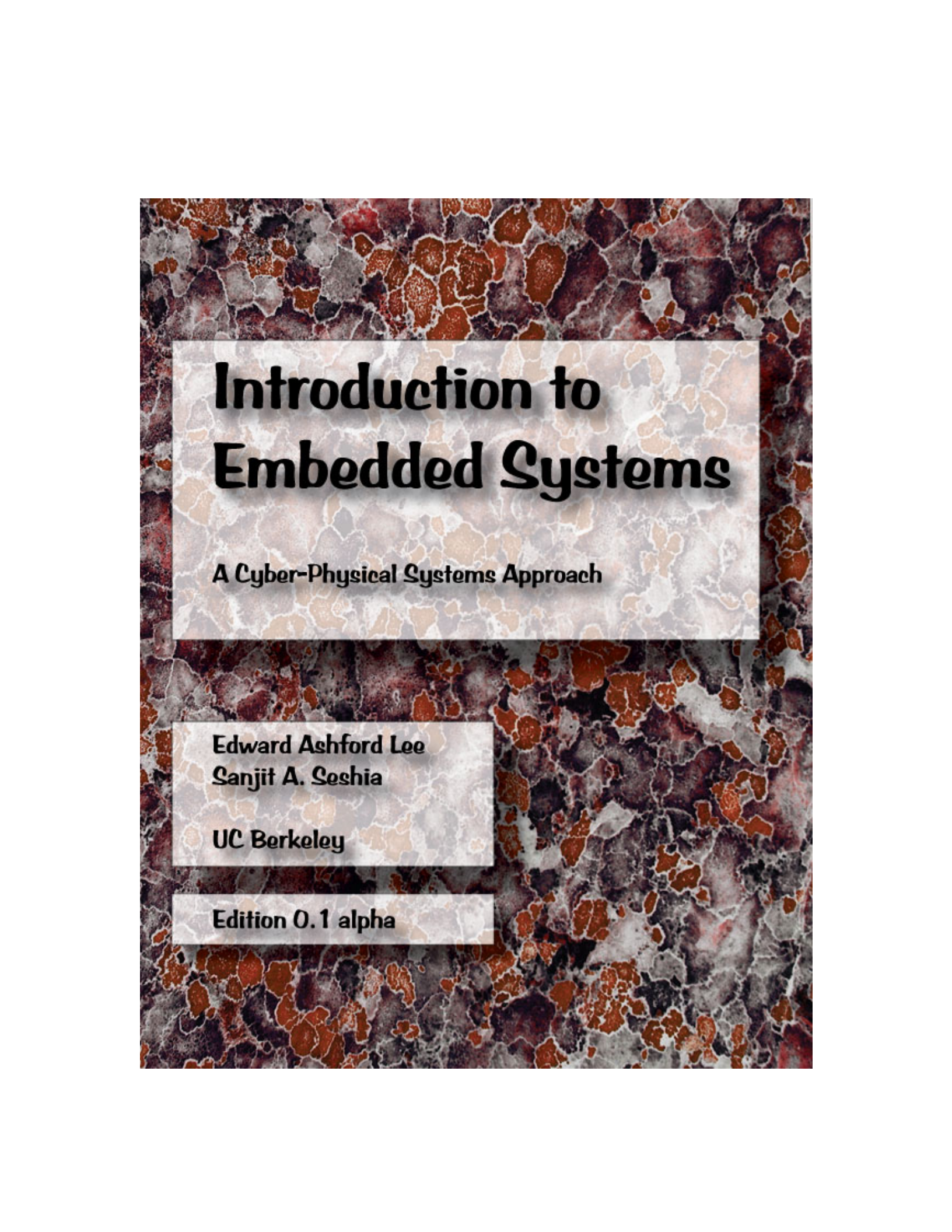




Introduction to Embedded Systems

A Cyber-Physical Systems Approach

**Edward Ashford Lee
Sanjit A. Seshia**

UC Berkeley

Edition 0.1 alpha

Copyright ©2009
Edward A. Lee & Sanjit A. Seshia
All rights reserved

DRAFT - Preliminary Release
January 20, 2010

Contents

Preface	ix
Notes to the Instructor	xiii
I Modeling Dynamic Behaviors	1
1 Continuous Dynamics	3
1.1 Newtonian Mechanics	4
1.2 Actor Models	8
1.3 Linearity and Time Invariance	11
1.4 Stability	12
1.5 Feedback Control	13
1.6 Summary	17
Exercises	19
2 Discrete Dynamics	21
2.1 Discrete Reactions	22
2.1.1 Input/Output Model	23
2.1.2 Denoting Inputs at a Reaction	24
2.2 The Notion of State	25
2.3 Finite-State Machines	26
2.3.1 Transitions	26
2.3.2 Implicit and Default Transitions	29
2.3.3 Update Functions	31
2.3.4 Stuttering	33
2.4 Nondeterminism	33
2.4.1 Uses of Non-Determinism	36
	iii

CONTENTS

2.5 Behaviors and Traces	36
Exercises	39
3 Hybrid Systems	43
3.1 Modal Models	44
3.1.1 Actor Model for State Machines	44
3.1.2 Continuous Inputs	45
3.1.3 State Refinements	46
3.2 Classes of Hybrid Systems	47
3.2.1 Timed Automata	47
3.2.2 Higher-Order Dynamics	50
3.2.3 Supervisory control	55
3.3 Summary	60
Exercises	62
4 Modeling Concurrent Systems	67
4.1 Synchronous Models	67
4.2 Time-Triggered Models	67
4.3 Event-Driven Models	67
4.4 Dataflow	67
4.5 Process Networks	68
II Design of Embedded Systems	69
5 Sensors and Actuators	71
5.1 Measuring Temperature	71
5.2 Measuring Motion	71
5.3 Measuring Time	72
5.4 Motion Control	72
5.4.1 Pulse Width Modulators	72
6 Embedded Processors	73
6.1 Types of Processors	74
6.1.1 Microcontrollers	74
6.1.2 Programmable DSPs	75

6.1.3	Graphics Processors	81
6.2	Parallelism	81
6.2.1	Parallelism vs. Concurrency	81
6.2.2	Instruction-Level Parallelism	83
6.2.3	Pipelining	88
6.2.4	Multicore Architectures	92
	Exercises	94
7	Memory Architectures	95
7.1	Volatility	96
7.1.1	Volatile Memory	96
7.1.2	Non-Volatile Memory	97
7.2	Memory Hierarchy	97
7.2.1	Memory Maps	98
7.2.2	Scratchpads and Caches	99
7.3	Memory Models	101
7.3.1	Memory Addresses	101
7.3.2	Stacks	102
7.3.3	Memory Protection Units	103
7.3.4	Dynamic Memory Allocation	103
7.3.5	Memory Model of C	104
8	Input and Output	105
8.1	I/O Hardware	106
8.1.1	General-Purpose Digital I/O	106
8.1.2	Serial Interfaces	110
8.1.3	Parallel Interfaces	113
8.2	Sequential Software in a Concurrent World	114
8.2.1	Interrupts and Exceptions	114
8.2.2	Atomicity	116
8.2.3	Interrupt Controllers	118
8.2.4	Modeling Interrupts	118
8.3	The Analog/Digital Interface	120
8.3.1	Digital to Analog and Analog to Digital Converters	121

CONTENTS

8.3.2	Signal Conditioning	122
8.3.3	Sampling and Aliasing	125
	Exercises	128
9	Multitasking	133
9.1	Processes	133
9.2	Threads	133
9.3	Message Passing	133
10	Networking and Distributed Systems	135
10.1	ISO Layers	136
10.2	Physical and Link Layer	136
10.3	Ethernet	136
10.4	CAN	136
10.5	FlexRay	136
10.6	Wireless	136
10.7	Time Synchronization	136
10.7.1	NTP	136
10.7.2	IEEE 1588	136
11	Scheduling	137
III	Analysis and Verification	139
12	Simulation	141
12.1	Efficient Random Sampling	141
12.2	Discrete Event Simulation	141
12.3	Numerical Simulation	141
12.4	Hardware-in-the-Loop Simulation	141
13	Invariants and Temporal Logic	143
13.1	Invariants	143
13.2	Safety and Liveness	143
13.3	Linear Temporal Logic	143
13.4	Other Temporal Logics	143

14	Equivalence and Refinement	145
14.1	Models as Specifications	145
14.2	Trace Equivalence and Containment	145
14.3	Simulation Relations	145
14.3.1	Relating behaviors	145
15	Reachability Analysis and Model Checking	147
15.1	Reachability and Invariants	147
15.2	Reachability Analysis	147
15.3	Model Checking	147
16	Quantitative Analysis	149
16.1	The Need for Quantitative Analysis	149
16.2	Execution Time Analysis	149
16.3	Memory Analysis	149
16.4	Power Analysis	149
IV	Appendices	151
A	Sets and Functions	153
A.1	Sets	153
A.2	Relations and Functions	154
A.2.1	Restriction and Projection	156
A.3	Sequences	157
	Exercises	158
	Bibliography	161
	Notation Index	164
	Index	167

CONTENTS

Preface

The most visible use of computers and software is information processing for human consumption. We use them to write books (like this one), search for information on the web, communicate via email, and keep track of financial data. The vast majority of computers in use, however, are much less visible. They run the engine, brakes, seatbelts, airbag, and audio system in your car. They digitally encode your voice and construct a radio signal to send it from your cell phone to a base station. They control your microwave oven, refrigerator, and dishwasher. They run printers ranging from desktop inkjet printers to large industrial high-volume printers. They command robots on a factory floor, power generation in a power plant, processes in a chemical plant, and traffic lights in a city. They search for microbes in biological samples, construct images of the inside of a human body, and measure vital signs. They process radio signals from space looking for supernovae and for extraterrestrial intelligence. They bring toys to life, enabling them to react to human touch and to sounds. They control aircraft and trains. These less visible computers are called **embedded systems**, and the software they run is called **embedded software**.

Despite this widespread prevalence of embedded systems, computer science has, throughout its relatively short history, focused primarily on information processing. Only recently have embedded systems received much attention from researchers. And only recently has the community recognized that the engineering techniques required to design and analyze these systems are distinct. Although embedded systems have been in use since the 1970s, for most of their history they were seen simply as small computers. The principal engineering problem was understood to be one of coping with limited resources (limited processing power, limited energy sources, small memories, etc.). As such, the engineering challenge was to optimize the designs. Since all designs benefit from optimization, the discipline was not distinct from anything else in computer science. It just had to be more aggressive about applying the same optimization techniques.

Recently, the community has come to understand that the principal challenges in embedded systems stem from their interaction with physical processes, and not from their limited resources. The term **cyber-physical systems (CPS)** was recently coined by Helen Gill at the National Science Foundation in the U.S. to refer to integrations of computation with physical processes. In CPS, embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. The design of such systems, therefore, requires understanding the joint dynamics of computers, software, networks, and physical

processes. It is this study of *joint* dynamics that sets this discipline apart.

When studying CPS, certain key problems emerge that are rare in so-called **general-purpose computing**. For example, in general-purpose software, the time it takes to perform a task is a performance issue, not a correctness issue. It is not incorrect to take longer to perform a task. It is merely less convenient and therefore less valuable. In CPS, the time it takes to perform a task may be critical to correct functioning of the system. In the physical world, as opposed to the cyber world, the passage of time is inexorable.

In CPS, moreover, many things happen at once. Physical processes are compositions of many things going on at once, unlike software processes, which are deeply rooted in sequential steps. [Abelson and Sussman \(1996\)](#) describe computing as “procedural epistemology,” knowledge through procedure. In the physical world, by contrast, processes are rarely procedural. Physical processes are compositions of many **concurrent** (from Latin, meaning “running together”) processes. Measuring and controlling the dynamics of these processes by orchestrating actions that influence the processes are the main tasks of embedded systems. Consequently, concurrency is intrinsic in CPS. Many of the technical challenges in designing and analyzing embedded software stem from the need to bridge an intrinsically sequential semantics with intrinsically concurrent physical world.

Today, getting computers to work together with physical processes requires technically intricate, low-level design. Embedded software designers are forced to struggle with interrupt controllers, memory architectures, assembly-level programming (to exploit specialized instructions or to precisely control timing), device driver design, network interfaces, scheduling strategies, etc., rather than focusing on specifying desired behavior. The sheer mass and complexity of these technologies tempts us to focus an introductory course on mastering them. But a better introductory course would focus on how to model and design the joint dynamics of software, networks, and physical processes. Such a course would present the technologies only as today’s (rather primitive) means of accomplishing those joint dynamics. This book is our attempt at a textbook for such a course.

Most texts on embedded systems focus on the collection of technologies needed to get computers to interact with physical systems ([Barr and Massa, 2006](#); [Berger, 2002](#); [Burns and Wellings, 2001](#); [Kamal, 2008](#); [Noergaard, 2005](#); [Parab et al., 2007](#); [Simon, 2006](#); [Valvano, 2007](#); [Wolf, 2000](#)). Others focus on adaptations of computer-science techniques (like programming languages, operating systems, networking, etc.) to dealing with technical problems in embedded systems ([Buttazzo, 2005](#); [Edwards, 2000](#); [Pottie and Kaiser, 2005](#)). While these implementation technologies are (today) necessary for system designers to get embedded systems working, they do not form the intellectual core of the discipline. The intellectual core is instead in models and abstractions that conjoin computation and physical dynamics.

A few textbooks offer efforts in this direction. [Jantsch \(2003\)](#) focuses on concurrent models of computation, [Marwedel \(2003\)](#) focuses on models of software and hardware behavior, and [Sriram and Bhattacharyya \(2009\)](#) focus on dataflow models of signal processing behavior and their mapping onto programmable DSPs. These are excellent starting points because models and concurrency (such as dataflow) and abstract models of software (such as statecharts) provide a far better starting point than imperative programming languages (like C), retrofits that provide concurrency on imper-

ative semantics (like interrupts and threads), and architectural annoyances that a designer must work around (like caches). These texts, however, are not suitable for an introductory course. They are either too specialized or too advanced or both. This book is our attempt to provide an introductory text that follows the spirit of focusing on models and their relationship to realizations of systems.

The major theme of this book is on models and their relationship to realizations of systems. The models we study are primarily about **dynamics**, the evolution of a system state in time. We do not address structural models, which represent static information about the construction of a system, although these too are important to embedded system design.

Working with models has a major advantage. Models can have formal properties. We can say definitive things about models. For example, we can assert that a model is **determinate**, meaning that given the same inputs it will always produce the same outputs. No such absolute assertion is possible with any physical realization of system. If our model is a good **abstraction** of the

Further Reading

Many textbooks on embedded systems have appeared in recent years. These books approach the subject in surprisingly diverse ways, often reflecting the perspective of a more established discipline that has migrated into embedded systems, such as VLSI design, control systems, signal processing, robotics, real-time systems, or software engineering. Some of these books complement the present one nicely. We strongly recommend them to the reader who wishes to broaden his or her understanding of the subject.

Specifically, [Patterson and Hennessey \(1996\)](#), although not focused on embedded processors, is the canonical reference for computer architecture, and a must-read for anyone interested embedded processor architectures. [Sriram and Bhattacharyya \(2009\)](#) focus on signal processing applications, such as wireless communications and digital media, and give particularly thorough coverage to [dataflow](#) programming methodologies. [Wolf \(2000\)](#) gives an excellent overview of hardware design techniques and microprocessor architectures and their implications for embedded software design. [Mishra and Dutt \(2005\)](#) give a view of embedded architectures based on architecture description languages (ADLs). [Oshana \(2006\)](#) specializes in **DSP** processors from Texas Instruments, giving an overview of architectural approaches and a sense of assembly-level programming. (p. 67)

Focused more on software, [Buttazzo \(2005\)](#) is an excellent overview of scheduling techniques for real-time software. [Liu \(2000\)](#) gives one of the best treatments yet of techniques for handling sporadic real-time events in software. [Edwards \(2000\)](#) gives a good overview of domain-specific higher-level programming languages used in some embedded system designs. [Pottie and Kaiser \(2005\)](#) give a good overview of networking technologies, particularly wireless, for embedded systems. (p. 77)

No single textbook can comprehensively cover the breadth of technologies available to the embedded systems engineer. We have found useful information in many of the books that focus primarily on today's design techniques ([Barr and Massa, 2006](#); [Berger, 2002](#); [Burns and Wellings, 2001](#); [Gajski et al., 2009](#); [Kamal, 2008](#); [Noergaard, 2005](#); [Parab et al., 2007](#); [Simon, 2006](#)).

physical system (meaning that omits only inessential details), then the definitive assertion about the model gives us confidence in the physical realization of the system. Such confidence is enormously valuable, particular for embedded systems where malfunctions can threaten human life. Studying models of systems gives us insight into how those systems will behave in the physical world.

Our focus is on the interplay of software and hardware with the physical environment in which they operate. This requires explicit modeling of the temporal dynamics of software and networks and explicit specification of concurrency properties intrinsic to the application. The fact that the implementation technologies have not yet caught up with this perspective should not cause us to teach the wrong engineering approach. We should teach design and modeling as it should be, and enrich this with a *critical* presentation of how to (partially) accomplish our objectives with today's technology. Embedded systems technologies today, therefore, should not be presented dispassionately as a collection of facts and tricks, as they are in many of the above cited books, but rather as stepping stones towards a sound design practice. The focus should be on what that sound design practice is, and on how today's technologies both impede and achieve it.

[Stankovic et al. \(2005\)](#) supports this view, stating that “existing technology for RTES [real-time embedded systems] design does not effectively support the development of reliable and robust embedded systems.” They cite a need to “raise the level of programming abstraction,” stating that “existing technology for RTES [real-time embedded systems] design does not effectively support the development of reliable and robust embedded systems.” We argue that raising the level of abstraction is insufficient. We have to also fundamentally change the abstractions that are used. Timing properties of software, for example, cannot be effectively introduced at higher levels of abstraction if they are entirely absent from the lower levels of abstraction on which these are built.

We require robust and predictable designs with repeatable temporal dynamics ([Lee, 2009](#)). We must do this by building abstractions that appropriately reflect the realities of cyber-physical systems. The result will be CPS designs that can be much more sophisticated, including more adaptive control logic, evolvability over time, improved safety and reliability, all without suffering from the **brittleness** of today's designs, where small changes have big consequences.

In addition to dealing with temporal dynamics, CPS designs invariably face challenging concurrency issues. Because software is so deeply rooted in sequential abstractions, concurrency mechanisms such as interrupts and multitasking, using semaphores and mutual exclusion, loom large. We therefore devote considerable effort in this book to developing a critical understanding of threads, message passing, deadlock avoidance, race conditions, and data determinism.

Notes to the Instructor

This book is intended for students at the advanced undergraduate level or the introductory graduate level, and for practicing engineers and computer scientists who wish to understand the engineering principles of embedded systems. We assume that the reader has some exposure to machine structures (e.g., should know what an ALU is and have some exposure to computer programming (we use C throughout the text), basic discrete mathematics and algorithms (e.g., should know basic graph theory, and algorithms for depth-first and breadth-first traversal) and at least an appreciation for signals and systems (what it means to sample a continuous-time signal, for example).

Part I

Modeling Dynamic Behaviors

Chapter 1

Continuous Dynamics

Contents

1.1	Newtonian Mechanics	4
1.2	Actor Models	8
1.3	Linearity and Time Invariance	11
1.4	Stability	12
1.5	Feedback Control	13
1.6	Summary	17
	Exercises	19

This chapter reviews a few of the many modeling techniques for studying dynamics of a physical system. A **system** is simply a combination of parts that is considered a whole. A **physical system** is one realized in matter, in contrast to a conceptual or logical system such as software and algorithms. The **dynamics** of a system is its evolution in time: how its state changes. A **model** of a system is a description of the system or of certain aspects of the system that is intended to yield insight into properties of the system. In this text, models have mathematical properties that enable systematic analysis. A major goal of this text is to develop an understanding of how to use models, of how to leverage their strengths and respect their weaknesses.

(p. xi)

A **cyber-physical system (CPS)** is system composed of physical subsystems together with computing and networking. Models of cyber-physical systems must include all three parts. The models will typically need to represent both **static properties** (those that do not change during the operation of the system) and dynamics. To lay the groundwork for CPS models, we begin with a review of models of the physical subsystems with an emphasis on dynamics.

(p. ix)

How to model physical systems is an enormous subject, much bigger than one chapter, or even one book. In fact, it is the focus of many branches of engineering, physics, chemistry, and biology. Our approach in this chapter is aimed at engineers. We assume some background in mathematical

modeling of dynamics, and then focus on how to compose diverse models. This will form the core of the cyber-physical system problem, since joint modeling of the cyber side, which is logical and conceptual, with the physical side, which is embodied in matter, is the core of the problem. We therefore make no attempt to be comprehensive, but rather pick a few modeling techniques that are widely used by engineers and well understood, review them, and then compose them together to form a cyber-physical whole.

We can begin by studying mechanical parts that move (this problem is known as **classical mechanics**). The techniques used to study the dynamics of such parts extend broadly to many other physical systems, including circuits, chemical processes, and biological processes. But mechanical parts are easiest for most people to visualize, so they make our example concrete. Motion of mechanical parts can often be modeled using **differential equations**, or equivalently, **integral equations**. Such models really only work well for “smooth” motion (a concept that we can make more precise using notions of linearity, time invariance, and continuity). For motions that are not smooth, such as those modeling collisions of mechanical parts, we can use modal models that represent distinct modes of operation with abrupt (conceptually instantaneous) transitions between modes. Collisions of mechanical objects can be useful modeled as discrete, instantaneous events. The problem of jointly modeling smooth motion and such discrete events is known as hybrid systems modeling and is studied in Chapter 3. Such combinations of discrete and continuous behaviors bring us one step closer to joint modeling of cyber and physical processes.

We begin with simple equations of motion, which provide a model of a system in the form of ordinary differential equations (**ODEs**). We then show how these ODEs can be represented in actor models, which include the class of models in popular modeling languages such as Simulink (from The MathWorks, Inc.). We then consider properties of such models such as linearity, time invariance, and stability, and consider consequences of these properties when manipulating models. We develop a simple example of a feedback control system that stabilizes an unstable system. Controllers for such systems are often realized using software, so such systems can serve as a canonical example of a cyber-physical system. The properties of the overall system emerge from properties of the cyber and physical parts.

1.1 Newtonian Mechanics

In this section, we give a brief working review of some principles of classical mechanics. This is intended to be just enough to be able to construct interesting models, but is by no means comprehensive. The interested reader is referred to many excellent texts on classical mechanics, including (Goldstein, 1980; Landau and Lifshitz, 1976; Marion and Thornton, 1995).

Motion in space of physical objects can be represented with **six degrees of freedom**, illustrated in Figure 1.1. Three of these represent position in three dimensional space, and three represent orientation in space. We assume three axes, x , y , and z , where by convention x is drawn increasing to the right, y is drawn increasing upwards, and z is drawn increasing out of the page. **Roll** θ_x

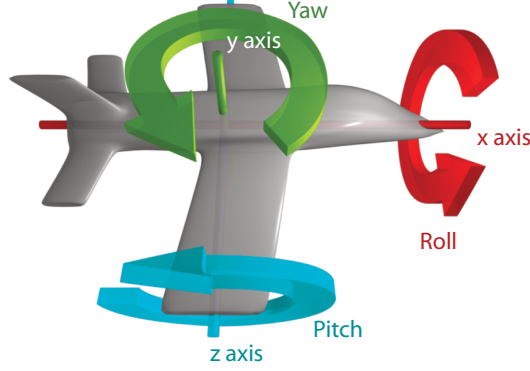


Figure 1.1: Modeling position with six degrees of freedom.

is an angle of rotation around the x axis, where by convention an angle of 0 radians represents horizontally flat along the z axis (i.e., the angle is given relative to the z axis). **Yaw** θ_y is the rotation around the y axis, where by convention 0 radians represents pointing directly to the right (i.e., the angle is given relative to the x axis). **Pitch** θ_z is rotation around the z axis, where by convention 0 radians represents pointing horizontally (i.e. the angle is given relative to the y axis).

The position of an object in space, therefore, is represented by six functions of the form $f: \mathbb{R} \rightarrow \mathbb{R}$, where the domain represents time and the codomain represents either distance along an axis or angle relative to an axis. Functions of this form are known as **continuous-time signals**. These are often collected into vector-valued functions $\mathbf{x}: \mathbb{R} \rightarrow \mathbb{R}^3$ and $\boldsymbol{\theta}: \mathbb{R} \rightarrow \mathbb{R}^3$, where $\mathbf{x}$ represents position, and $\boldsymbol{\theta}$ represents orientation.

Changes in position or orientation are governed by **Newton's second law**, relating force with acceleration. Acceleration is the second derivative of position. Our first equation handles the position information,

$$\mathbf{F}(t) = M\ddot{\mathbf{x}}(t), \quad (1.1)$$

where $\mathbf{F}$ is the force vector in three directions, M is the mass of the object, and $\ddot{\mathbf{x}}$ is the second derivative of $\mathbf{x}$ with respect to time (i.e. the acceleration). Velocity is the integral of acceleration, given by

$$\forall t > 0, \quad \dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \int_0^t \ddot{\mathbf{x}}(\tau) d\tau$$

where $\dot{\mathbf{x}}(0)$ is the initial velocity in three directions. Using (1.1), this becomes

$$\forall t > 0, \quad \dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M} \int_0^t \mathbf{F}(\tau) d\tau,$$

Position is the integral of velocity,

$$\begin{aligned}\mathbf{x}(t) &= \mathbf{x}(0) + \int_0^t \dot{\mathbf{x}}(\tau) d\tau \\ &= \mathbf{x}(0) + t\dot{\mathbf{x}}(0) + \frac{1}{M} \int_0^t \int_0^\tau \mathbf{F}(\alpha) d\alpha d\tau,\end{aligned}$$

where $\mathbf{x}(0)$ is the initial position. Using these equations, if you know the initial position and initial velocity of an object and the forces on the object in all three directions as a function of time, you can determine the acceleration, velocity, and position of the object at any time.

The versions of these motions of equation that affect orientation use **torque**, the rotational version of force. It is again a three-element vector as a function of time, representing the net twisting force on an object. It can be related to angular velocity in a manner similar to (1.1),

$$\mathbf{T}(t) = \frac{d}{dt} (\mathbf{I}(t)\dot{\boldsymbol{\theta}}(t)), \quad (1.2)$$

where $\mathbf{T}$ is the torque vector in three axes and $\mathbf{I}(t)$ is the **moment of inertia tensor** of the object. The moment of inertia is a 3×3 matrix that depends on the geometry and orientation of the object. Intuitively, it represents the reluctance that an object has to spin around any axis as a function of its orientation along the three axes. If the object is spherical, for example, this reluctance is the same around all axes, so it reduces to a constant scalar I (or equivalently, to a diagonal matrix $\mathbf{I}$ with equal diagonal elements I). The equation then looks much more like (1.1),

$$\mathbf{T}(t) = I\ddot{\boldsymbol{\theta}}(t). \quad (1.3)$$

To be explicit about the three dimensions, we might write (1.2) as

$$\begin{bmatrix} T_x(t) \\ T_y(t) \\ T_z(t) \end{bmatrix} = \frac{d}{dt} \left(\begin{bmatrix} I_{xx}(t) & I_{xy}(t) & I_{xz}(t) \\ I_{yx}(t) & I_{yy}(t) & I_{yz}(t) \\ I_{zx}(t) & I_{zy}(t) & I_{zz}(t) \end{bmatrix} \begin{bmatrix} \dot{\theta}_x(t) \\ \dot{\theta}_y(t) \\ \dot{\theta}_z(t) \end{bmatrix} \right).$$

Here, for example, $T_y(t)$ is the net torque around the y axis (which would cause changes in yaw), $I_{yx}(t)$ is the inertia that determines how acceleration around the x axis is related to torque around the y axis.

Rotational velocity is the integral of acceleration,

$$\dot{\boldsymbol{\theta}}(t) = \dot{\boldsymbol{\theta}}(0) + \int_0^t \ddot{\boldsymbol{\theta}}(\tau) d\tau,$$

where $\dot{\boldsymbol{\theta}}(0)$ is the initial rotational velocity in three axes. For a spherical object, using (1.3), this becomes

$$\dot{\boldsymbol{\theta}}(t) = \dot{\boldsymbol{\theta}}(0) + \frac{1}{I} \int_0^t \mathbf{T}(\tau) d\tau.$$

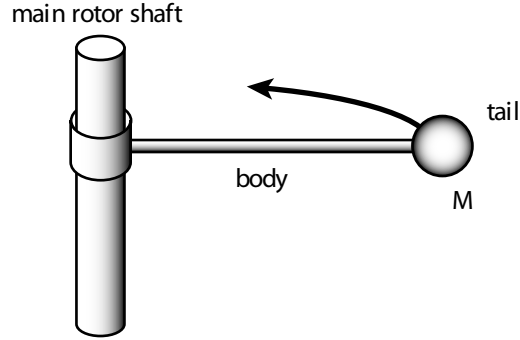


Figure 1.2: Simplified model of a helicopter.

Orientation is the integral of rotational velocity,

$$\begin{aligned}\theta(t) &= \theta(0) + \int_0^t \dot{\theta}(\tau) d\tau \\ &= \theta(0) + t\dot{\theta}(0) + \frac{1}{I} \int_0^t \int_0^\tau \mathbf{T}(\alpha) d\alpha d\tau\end{aligned}$$

where $\theta(0)$ is the initial orientation. Using these equations, if you know the initial orientation and initial rotational velocity of an object and the torques on the object in all three axes as a function of time, you can determine the rotational acceleration, velocity, and orientation of the object at any time.

Often, as we have done for a spherical object, we can simplify by reducing the number of dimensions that are considered. In general, such a simplification is called a **model-order reduction**. For example, if an object is a moving vehicle on a flat surface, there may be little reason to consider the y axis movement or the pitch or roll of the object.

Example 1.1: Consider a simple control problem that admits such reduction of dimensionality. A helicopter has two rotors, one above, which provides lift, and one on the tail. Without the rotor on the tail, the body of the helicopter would start to spin. The rotor on the tail counteracts that spin. Specifically, the force produced by the tail rotor must perfectly counter the friction with the main rotor, or the body will spin. Here we consider this role of the tail rotor independently from all other motion of the helicopter.

A highly simplified version of the helicopter is shown in figure 1.2. In this version, we assume that the helicopter position is fixed at the origin, and hence there is no need to consider the equations governing the dynamics of position. Moreover, we will assume that the helicopter remains vertical, so pitch and roll are fixed at zero. Note that these

assumptions are not as unrealistic as they may seem since we can define the coordinate system to be fixed to the helicopter.

With these assumptions, the moment of inertia reduces to a scalar that resists changes in yaw. The torque causing changes in yaw will be due to the friction with the main rotor. This will tend to cause the helicopter to rotate in the same direction as the rotor rotation. The tail rotor has the job of countering that torque to keep the body of the helicopter from spinning.

We model the simplified helicopter by a system that takes as input a continuous-time signal T_y , the torque around the y axis (which causes changes in yaw). This torque is the net difference between that caused by the friction of the main rotor and that caused by the tail rotor. The output of our system will be the angular velocity $\dot{\theta}_y$ around the y axis. The dimensionality-reduced version of (1.2) can be written as

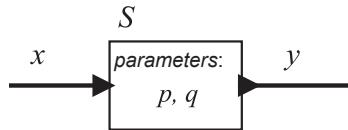
$$\ddot{\theta}_y(t) = T_y(t)/I_{yy}.$$

Integrating both sides, we get the output $\dot{\theta}$ as a function of the input T_y ,

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau. \quad (1.4)$$

1.2 Actor Models

In the previous section, a model of a physical system is given by a differential or an integral equation that relates input signals (force or torque) to output signals (position, orientation, velocity, or rotational velocity). Such a physical system can be viewed as a component in a larger system. In particular, a continuous-time system may be modeled by a box with an input **port** and an output port as follows:



where the input signal x and the output signal y are functions of the form

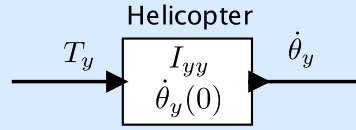
$$x: \mathbb{R} \rightarrow \mathbb{R}, \quad y: \mathbb{R} \rightarrow \mathbb{R}.$$

Here the domain represents time and the codomain represents the value of the signal at a particular time. The model itself is a function of the form

$$S: X \rightarrow Y, \quad (1.5)$$

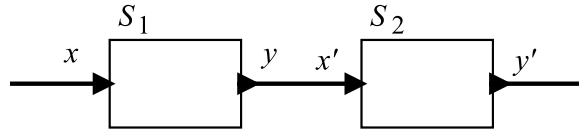
where $X = Y = \mathbb{R}^{\mathbb{R}}$, the set of functions that map the reals into the reals, like x and y above.¹ The function S may depend on parameters of the system, in which case the parameters may be optionally shown in the box, and may be optionally included in the function notation. For example, in the above figure, if there are parameters p and q , we might write the system function as $S_{p,q}$ or even $S(p,q)$, keeping in mind that both of notations represent functions of the form in 1.5. A box like that above, where the inputs are functions and the outputs are functions, is called an **actor**.

Example 1.2: The actor model for the helicopter of example 1.1 can be depicted as follows:



The input and output are both continuous-time functions. The parameters of the actor are the initial angular velocity $\dot{\theta}_y(0)$ and the moment of inertia I_{yy} . The function of the actor is defined by (1.4).

Actor models are composable. In particular, given two actors S_1 and S_2 , we can form a **cascade composition** as follows:

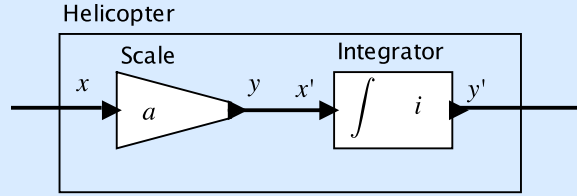


In the diagram, the “wire” between the output of S_1 and the input of S_2 means precisely that $y = x'$, or more pedantically,

$$\forall t \in \mathbb{R}, \quad y(t) = x'(t).$$

¹As explained in Appendix A, the notation $\mathbb{R}^{\mathbb{R}}$ (which can also be written $(\mathbb{R} \rightarrow \mathbb{R})$) represents the set of all functions with domain $\mathbb{R}$ and codomain $\mathbb{R}$.

Example 1.3: The actor model for the helicopter can be represented as a cascade composition of two actors as follows:



The left actor represents a simple scaling function parameterized by the constant a defined by

$$\forall t \in \mathbb{R}, \quad y(t) = ax(t). \quad (1.6)$$

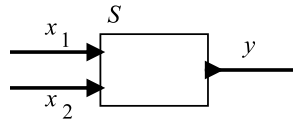
More compactly, we can write $y = ax$, where it is understood that the product of a scalar a and a function x is interpreted as in (1.6). The right actor represents an integrator parameterized by the initial value i defined by

$$\forall t \in \mathbb{R}, \quad y'(t) = i + \int_0^t x'(\tau) d\tau.$$

If we give the parameter values $a = 1/I_{yy}$ and $i = \dot{\theta}_y(0)$, we see that this system represents (1.4) where the input $x = T_y$ is torque and the output $y' = \dot{\theta}_y$ is angular velocity.

In the above figure, we have customized the **icons**, which are the boxes representing the actors. These particular actors (scaler and integrator) are particularly useful building blocks for building up models of physical dynamics, so assigning them recognizable visual notations is useful.

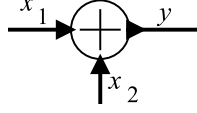
We can have actors that have multiple input signals and/or multiple output signals. These are represented similarly, as in the following example which has two input signals and one output signal:



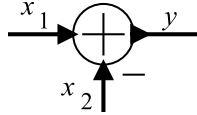
A particularly useful building block with this form is a signal **adder**, defined by

$$\forall t \in \mathbb{R}, \quad y(t) = x_1(t) + x_2(t).$$

This will often be represented by a custom icon as follows:



Sometimes, one of the inputs will be subtracted rather than added, in which case the icon is further customized with minus sign near that input, as below:



This actor represents a function $S: (\mathbb{R} \rightarrow \mathbb{R})^2 \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$ given by

$$\forall t \in \mathbb{R}, \forall x_1, x_2 \in (\mathbb{R} \rightarrow \mathbb{R}), \quad (S(x_1, x_2))(t) = y(t) = x_1(t) - x_2(t).$$

Notice the careful notation. $S(x_1, x_2)$ is a function in $\mathbb{R}^{\mathbb{R}}$. Hence, it can be evaluated at a $t \in \mathbb{R}$.

In the rest of this chapter, we will not make a distinction between a system and its actor model, unless the distinction is essential to the argument. We will assume that the actor model captures everything of interest about the system. This is an admittedly bold assumption, since generally the properties we assume are genuinely properties of the actor model, but only approximate descriptions of the actual system.

1.3 Linearity and Time Invariance

Systems that are linear and time invariant (LTI) have particularly nice mathematical properties. Much of the theory of control systems depends on these properties. These properties form the main body of courses on signals and systems, and are beyond the scope of this text. But we will occasionally exploit simple versions of the properties, so it is useful to determine when a system is LTI.

A system $S: X \rightarrow Y$, where X and Y are sets of signals, is linear if it satisfies the **superposition** property:

$$\forall x_1, x_2 \in X \text{ and } \forall a, b \in \mathbb{R}, \quad S(ax_1 + bx_2) = aS(x_1) + bS(x_2).$$

It is easy to see that the helicopter system defined in example 1.1 is linear if and only if the initial angular velocity $\dot{\theta}_y(0) = 0$ (see exercise 1).

More generally, it is easy to see that an integrator as defined in example 1.3 is linear if and only if the initial value $i = 0$, that the Scale actor is always linear, and that the cascade of any two linear actors is linear. We can trivially extend the definition of linearity to actors with more than one input or output signal and then determine that the adder is also linear.

To define time invariance, we first define a specialized continuous-time actor called a **delay**. Let $D_\tau: X \rightarrow Y$, where X and Y are sets of continuous-time signals, be defined by

$$\forall x \in X \text{ and } \forall t \in \mathbb{R}, \quad (D_\tau(x))(t) = x(t - \tau). \quad (1.7)$$

Here, τ is a parameter of the delay actor. A system $S: X \rightarrow Y$ is time invariant if

$$\forall x \in X \text{ and } \forall \tau \in \mathbb{R}, \quad S(D_\tau(x)) = D_\tau(S(x)).$$

The helicopter system defined in example 1.1 and (1.4) is not time-invariant. A minor variant, however, is:

$$\dot{\theta}_y(t) = \frac{1}{I_{yy}} \int_{-\infty}^t T_y(\tau) d\tau.$$

This version does not allow for an initial angular rotation.

A **linear time-invariant system (LTI)** is a system that is both linear and time invariant. A major objective in modeling physical dynamics is to choose an LTI model whenever possible. If reasonable approximations result in an LTI model, it is worth making these approximations. It is not always easy to determine whether the approximations are reasonable, or to find models for which the approximation is reasonable. It is often easy to construct models that are more complicated than they need to be (see exercise 2).

1.4 Stability

A system is said to be **bounded-input bounded-output stable (BIBO stable or just stable)** if the output signal is bounded for all input signals that are bounded.

Consider a continuous-time system with input w and output v . The input is bounded if there is a real number $A < \infty$ such that $|w(t)| \leq A$ for all $t \in \mathbb{R}$. The output is bounded if there is a real number $B < \infty$ such that $|v(t)| \leq B$ for all $t \in \mathbb{R}$. The system is stable if for any input bounded by A , there is some bound B on the output.

Example 1.4: It is now easy to see that the helicopter system developed in example 1.1 is unstable. Let the input be $T_y = u$, where u is the **unit step**, given by

$$\forall t \in \mathbb{R}, \quad u(t) = \begin{cases} 0, & t < 0 \\ 1, & t \geq 0 \end{cases}. \quad (1.8)$$

This means that prior to time zero, there is no torque applied to the system, and starting at time zero, we apply a torque of unit magnitude. This input is clearly bounded. It never exceeds one in magnitude. However, the output grows without bound.

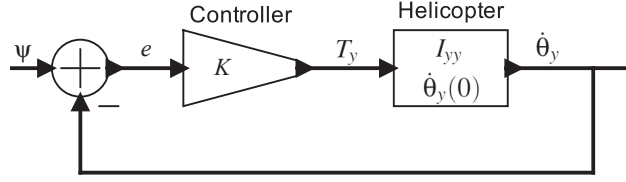


Figure 1.3: Proportional control system that stabilizes the helicopter.

In practice, a helicopter uses a feedback system to determine how much torque to apply at the tail rotor to keep the body of the helicopter straight. We study how to do that next.

1.5 Feedback Control

Feedback control is a sophisticated topic, easily occupying multiple texts and complete courses. Here, we only barely touch on the subject, just enough to motivate the interactions between software and physical systems. Feedback control systems are often implemented using embedded software, and the overall physical dynamics is a composition of the software and physical dynamics. More detail can be found in chapters 12-14 of [Lee and Varaiya \(2003\)](#).

Example 1.5: Recall that the helicopter model of example 1.1 is not stable. We can stabilize it with a simple feedback control system, as shown in figure 1.3. The input ψ to this system is a continuous-time system specifying the desired angular velocity. The **error signal** e represents the difference between the actual and the desired angular velocity. In the figure, the controller simply scales the error signal by a constant K , providing a control input to the helicopter. We use (1.4) to write

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau \quad (1.9)$$

$$= \dot{\theta}_y(0) + \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau, \quad (1.10)$$

where we have used the facts (from the figure),

$$e(t) = \psi(t) - \dot{\theta}_y(t)$$

and

$$T_y(t) = Ke(t).$$

Equation (1.10) has $\dot{\theta}_y(t)$ on both sides, and therefore is not trivial to solve. The easiest solution technique uses Laplace transforms (see Lee and Varaiya (2003) chapter 14). However, for our purposes here, we can use a more brute-force technique from calculus. To make this as simple as possible, we assume that $\psi(t) = 0$ for all t ; i.e., we wish to control the helicopter simply to keep it from rotating at all. The desired angular velocity is zero. In this case, (1.10) simplifies to

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau. \quad (1.11)$$

Using the fact from calculus that

$$\int_0^t a e^{a\tau} d\tau = 1 - e^{at} u(t),$$

where u is given by (1.8), we can infer that the solution to (1.11) is

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) e^{-Kt/I_{yy}} u(t). \quad (1.12)$$

(Note that although it is easy to verify that this solution is correct, deriving the solution is not so easy. For this purpose, Laplace transforms provide a far better mechanism.)

We can see from (1.12) that the angular velocity approaches the desired angular velocity (zero) as t gets large as long as K is positive. For larger K , it will approach more quickly. For negative K , the system is unstable, and angular velocity will grow without bound.

The previous example illustrates a **proportional control** feedback loop. It is called this because the control signal is proportional to the error. We assumed a desired signal of zero. It is equally easy to assume that the helicopter is initially at rest (the angular velocity is zero) and then determine the behavior for a particular non-zero desired signal, as we do in the following example.

Example 1.6: Assume that helicopter is **initially at rest**,

$$\dot{\theta}(0) = 0,$$

and that the desired signal is

$$\psi(t) = au(t)$$

for some constant a . That is, we wish to control the helicopter to get it to rotate at a fixed rate.

We use (1.4) to write

$$\begin{aligned}
 \dot{\theta}_y(t) &= \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau \\
 &= \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau \\
 &= \frac{K}{I_{yy}} \int_0^t a d\tau - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau \\
 &= \frac{Kat}{I_{yy}} - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau.
 \end{aligned}$$

Using the same (black magic) technique of inferring and then verifying the solution, we can see that the solution is

$$\dot{\theta}_y(t) = au(t)(1 - e^{-Kt/I_{yy}}). \quad (1.13)$$

Again, the angular velocity approaches the desired angular velocity as t gets large as long as K is positive. For larger K , it will approach more quickly. For negative K , the system is unstable, and angular velocity will grow without bound.

Note that the first term in the above solution is exactly the desired angular velocity. The second term is an error called the **tracking error**, that for this example asymptotically approaches zero.

The above example is somewhat unrealistic because we cannot independently control the *net* torque of the helicopter. In particular, the net torque T_y is the sum of the torque T_t due to the friction of the top rotor and the torque T_r due to the tail rotor,

$$\forall t \in \mathbb{R}, \quad T_y(t) = T_t(t) + T_r(t).$$

T_t will be determined by the rotation required to maintain or achieve a desired altitude, quite independent of the rotation of the helicopter. Thus, we will actually need to design a control system that controls T_r and stabilizes the helicopter for any T_t (or, more precisely, any T_t within operating parameters). In the next example, we study how this changes the performance of the control system.

Example 1.7: In Figure 1.4(a), we have modified the helicopter model so that it has two inputs, T_t and T_r , the torque due to the top rotor and tail rotor respectively. The feedback

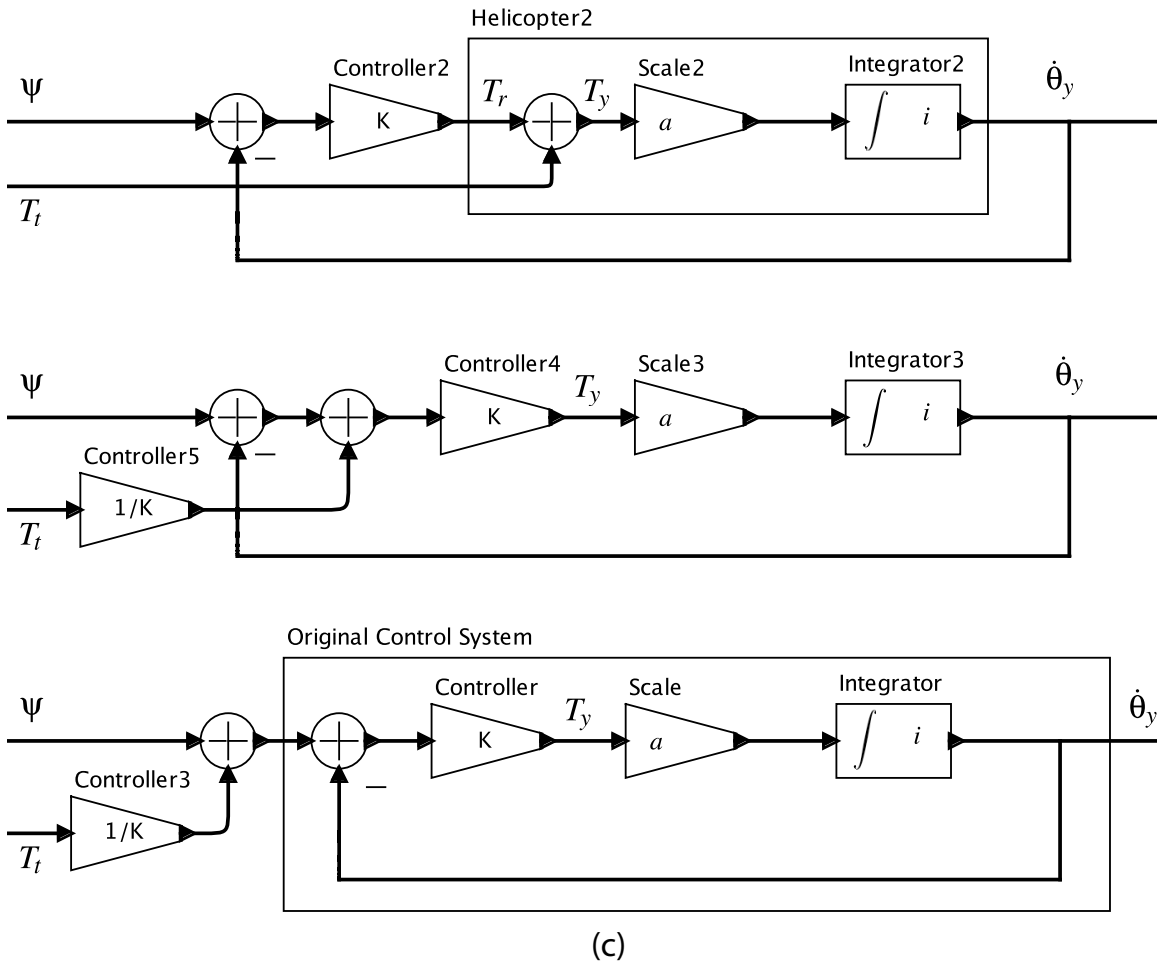


Figure 1.4: (a) Helicopter model with separately controlled torques for the top and tail rotors. (b) Transformation to an equivalent model. (c) Further transformation to an equivalent model that we can use to understand the behavior of the controller.

control system is now controlling only T_r , and T_t is treated as an external (uncontrolled) input signal. How well will this control system behave?

Again, a full treatment of the subject is beyond the scope of this text, but we will study a specific example. Suppose that the torque due to the top rotor is given by

$$T_t = bu(t)$$

for some constant b . That is, at time zero, the top rotor starts spinning a constant velocity, and then holds that velocity. Suppose further that the helicopter is initially at rest. We can use the results of Example 1.6 to find the behavior of the system.

First, we transform the model into the equivalent model shown in Figure 1.4(b). This transformation simply relies on the algebraic fact that for any real numbers a_1, a_2, K ,

$$Ka_1 + a_2 = K(a_1 + a_2/K).$$

We further transform the model to get the equivalent model shown in Figure 1.4(c), which has used the fact that addition is commutative. In Figure 1.4(c), we see that the portion of the model enclosed in the box is exactly the same as the control system analyzed in Example 1.6, shown in Figure 1.3. Thus, the same analysis as in Example 1.6 still applies. Suppose that desired angular rotation is

$$\psi(t) = 0.$$

Then the input to the original control system will be

$$x(t) = \psi(t) + T_t(t)/K = (b/K)u(t).$$

From (1.13), we see that the solution is

$$\dot{\theta}_y(t) = (b/K)u(t)(1 - e^{-Kt/I_{yy}}). \quad (1.14)$$

The desired angular rotation is zero, but the control system asymptotically approaches a non-zero angular rotation of b/K . This tracking error can be made arbitrarily small by increasing the control system feedback gain K , but with this controller design, it cannot be made to go to zero. An alternative controller design that yields an asymptotic tracking error of zero is studied in Exercise 4.

1.6 Summary

This chapter has described two distinct modeling techniques that describe physical dynamics. The first is ordinary differential equations, a venerable toolkit for engineers, and the second is actor

1.6. SUMMARY

models, a newer technique driven by software modeling and simulation tools. The two are closely related. This chapter has emphasized the relationship between these models, and the relationship of those models to the systems being modeled. These relationships, however, are quite a deep subject that we have barely touched upon. Our objective is to focus the attention of the reader on the fact that distinct models are distinct, and that models are distinct from the systems being modeled. The **fidelity** of a model (how well it approximates the system being modeled) is a strong factor in the success or failure of any engineering effort.

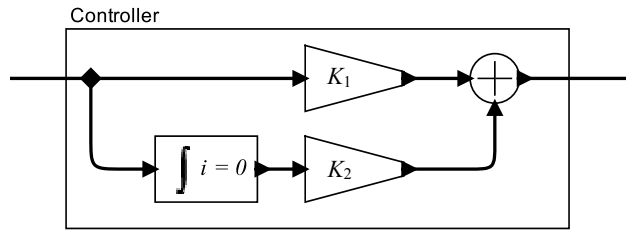


Figure 1.5: A PI controller for the helicopter.

Exercises

1. This exercise studies linearity.
 - (a) Show that the helicopter system defined in example 1.1 is linear if and only if the initial angular velocity $\dot{\theta}_y(0) = 0$.
 - (b) Show that the cascade of any two linear actors is linear.
 - (c) Augment the definition of linearity so that it applies to actors with two input signals and one output signal. Show that the adder actor is linear.
2. Consider the helicopter of example 1.1, but with a slightly different definition of the input and output. Suppose that, as in the example, the input is $T_y: \mathbb{R} \rightarrow \mathbb{R}$, as in the example, but the output is the position of the tail relative to the main rotor shaft. Is this an LTI system? Is this a BIBO stable system?
3. Consider a rotating robot where you can control the angular velocity around a fixed axis.
 - (a) Model this as a system where the input is angular velocity $\dot{\theta}$ and the output is angle θ . Give your model as an equation relating the input and output as functions of time.
 - (b) Is this system BIBO stable?
 - (c) Design a proportional controller to set the robot onto a desired angle. That is, assume that the initial angle is $\theta(0) = 0$, and let the desired angle be $\psi(t) = au(t)$. Find the actual angle as a function of time and the proportional controller feedback gain K . What is your output at $t = 0$? What does it approach as t gets large?
4. (a) Using your favorite continuous-time modeling software (such as LabVIEW, Simulink, or Ptolemy II), construct a model of the helicopter control system shown in Figure 1.4. Choose some reasonable parameters and plot the actual angular velocity as a function of time, assuming that the desired angular velocity is zero, $\psi(t) = 0$, and that the top-rotor torque is non-zero, $T_t(t) = bu(t)$. Give your plot for several values of K and discuss how the behavior varies with K .

EXERCISES

- (b) Modify the model of part (a) to replace the Controller of Figure 1.4 (the simple scale-by- K actor) with the alternative controller shown in figure 1.5. This alternative controller is called a **proportional-integrator (PI) controller**. It has two parameter K_1 and K_2 . Experiment with the values of these parameters, give some plots of the behavior with the same inputs as in part (a), and discuss the behavior of this controller in contrast to the one of part (a).

Chapter 2

Discrete Dynamics

Contents

2.1	Discrete Reactions	22
2.1.1	Input/Output Model	23
2.1.2	Denoting Inputs at a Reaction	24
2.2	The Notion of State	25
2.3	Finite-State Machines	26
2.3.1	Transitions	26
2.3.2	Implicit and Default Transitions	29
2.3.3	Update Functions	31
2.3.4	Stuttering	33
2.4	Nondeterminism	33
2.4.1	Uses of Non-Determinism	36
2.5	Behaviors and Traces	36
	Exercises	39

Models of embedded systems include both **discrete** and **continuous** components. Loosely, continuous components evolve smoothly, while discrete components evolve abruptly. The previous chapter considered continuous components, and showed that the physical dynamics of the system can often be modeled with ordinary differential or integral equations, or equivalently with actor models that mirror these equations. Discrete components, on the other hand, are not conveniently modeled by ODEs. In this chapter, we study how state machines can be used to model discrete dynamics. In the next chapter, we will show how these state machines can be combined with models of continuous dynamics to get hybrid system models.

(p. 4)

2.1 Discrete Reactions

Some systems are inherently discrete.

Example 2.1: Consider a system that counts the number of cars that enter and leave a parking garage in order to keep track of how many cars are in the garage at any time. It could be implemented as shown in Figure 2.1. We ignore for now how to design the sensors that detect the entry or departure of cars (that problem is considered in Chapter 5). We simply assume that the *ArrivalDetector* actor produces an event when a car arrives, and the *DepartureDetector* actor produces an event when a car departs. The *Counter* actor keeps a running count, starting from an initial value i . Each time the count changes, it produces an output event that updates a display.

In the above example, each entry or departure is modeled as a **discrete event**. A discrete event occurs at an instant of time rather than over time. The *Counter* actor in Figure 2.1 is analogous to the integrator actor used in the previous chapter, shown here in Figure 2.2. Like the counter actor, the integrator accumulates input values. However, it does so very differently. The input of an integrator is a function of the form $x: \mathbb{R} \rightarrow \mathbb{R}$, a continuous-time signal. The *up* input of the counter, on the other hand, can be modeled as a function of the form

$$up: \mathbb{R} \rightarrow \{absent, present\}.$$

This means that at any time $t \in \mathbb{R}$, the input $up(t)$ is either *absent*, meaning that there is no event at that time, or *present*, meaning that there is. A signal of this form is known as a **pure signal**. It carries no value, but instead provides all its information by either present or absent at any given time. The *down* and *count* signals in Figure 2.1 are similarly modeled as pure signals.

Assume our counter operates as follows. When an event is present at the *up* input, it increments its count and produces on the output the new value of the count. When an event is present at the *down*

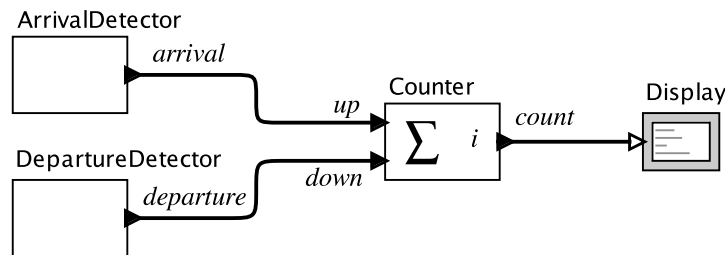


Figure 2.1: Model of a system that keeps track of the number of cars in a parking garage.

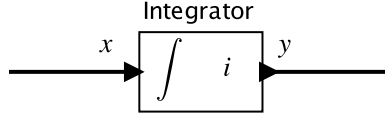


Figure 2.2: Icon for the integrator actor used in the previous chapter.

input, it decrements its count and produces on the output the new value of the count.¹ At all other times (when both inputs are absent), it produces no output (the *count* output is absent). Assume further that the inputs are absent most of the time, or more technically, that the inputs are discrete (see the sidebar on page 25). Then the counter reacts in sequence to each of a sequence of input events. This is very different from the integrator, which reacts continuously a continuum of inputs.

2.1.1 Input/Output Model

A system that operates in a sequence of discrete **reactions** is said to have **discrete dynamics**. Its dynamics can be described as a sequence of steps that we call reactions, each of which we assume to be instantaneous. Let us characterize the reactions of the counter and compare them to the integrator.

The integrator is a function of the form

$$\text{Integrator}: \mathbb{R}^{\mathbb{R}} \rightarrow \mathbb{R}^{\mathbb{R}},$$

which can also be written²

$$\text{Integrator}: (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R}).$$

This just tells us that the input and output are continuous-time signals. Correspondingly, the counter is a function of the form

$$\text{Counter}: (\mathbb{R} \rightarrow \{\text{absent}, \text{present}\})^P \rightarrow (\mathbb{R} \rightarrow \{\text{absent}\} \cup \mathbb{N}), \quad (2.1)$$

where $\mathbb{N}$ is the natural numbers and P is the set of input ports,

(p. 8)

$$P = \{\text{up}, \text{down}\}. \quad (2.2)$$

Recall that the notation A^B denotes the set of all functions from B to A . Hence, the input to the function *Counter* is a function whose domain is P that for each port $p \in P$ yields a function in $(\mathbb{R} \rightarrow \{\text{absent}, \text{present}\})$. That latter function, in turn, for each time $t \in \mathbb{R}$ yields either *absent* or *present*.

The input to the counter is a pair of discrete signals that at certain times have an event (are present), and at other times have no event. The output also a discrete signal that, when an input is present,

¹It would be wise to design this system with a fault handler that does something reasonable if the count drops below zero, but we ignore this for now.

²See Appendix A if the notation is unfamiliar.

has a value that is a natural number, and at other times is absent.³ Clearly, there is no need for this counter to do anything when the input is absent. It only needs to operate when inputs are present. Hence, it has discrete dynamics.

2.1.2 Denoting Inputs at a Reaction

A particular reaction will observe the values of the inputs at a particular time t and calculate output values for that same time t . It is useful therefore to come up with a concise notation to describe all the inputs at time t , rather than the notation in (2.1), which describes the inputs over time. For the counter actor, at time t , we can model the input as a function of the form $i: P \rightarrow \{absent, present\}$, where P is given by (2.2). For example, if a car is arriving at time t and no car is departing, then $i(up) = present$ and $i(down) = absent$.

In general, we denote the set of all possible input values when a reaction occurs by *Inputs* and the set of all possible output values is denoted by *Outputs*. For the counter,

$$Inputs = (\{up, down\} \rightarrow \{absent, present\})$$

and

$$Outputs = (\{count\} \rightarrow \{absent\} \cup \mathbb{N}),$$

where *count* is the name of the output port. Notice that for the output, since there is only one output port, there is no real need to distinguish between output ports. But for the inputs, we do have to be able to distinguish which of the two inputs is present, *up* or *down*. The structure of these sets supports any number of input or output ports.

(p. 153) For pure signals, the above notation is rather pedantic and can unnecessarily obscure what is going on. Hence, for pure signals, we can use a simple shorthand. Suppose that a set P of input ports have only pure signals. Then we can let $Inputs = 2^P$, the powerset of P (set of all subsets). Then we can denote the inputs at time t by a subset $i \in Inputs$, or equivalently, $i \subseteq P$. The subset i will include only those ports whose signals are present at t .

Example 2.2: For the garage counter, $P = \{up, down\}$. If a car is arriving at time t and none is departing, the input is $i = \{up\}$. If a car is arriving and another is departing at the same time, we say that the input is $i = \{up, down\}$. If neither is true, then we say $i = \emptyset$, the empty set.

³As shown in Exercise 5, the fact that input signals are discrete does not necessarily imply that the output signal is discrete. However, for this application, there are physical limitations on the rates at which cars can arrive and depart that ensure that these signals are discrete. So it is safe to assume that they are discrete.

This notation, of course, can only be used when we can be sure we are not creating confusion between the function $up: \mathbb{R} \rightarrow \{absent, present\}$ and the name up of the input port. For pure signals, this confusion is not likely to occur.

2.2 The Notion of State

Intuitively, the **state** of a system or subsystem is its condition at a particular point in time. In general, the state affects how the (sub)system reacts to inputs. Formally, we define the state to be an encoding of everything about the past that has an effect on the system's reaction to current or future inputs. The state is a summary of the past.

Consider the integrator actor shown in Figure 2.2. This actor has state, which in this case happens to have the same value as the output at any time t . The state of the actor at a time t is the value of the integral of the input signal up to time t . In order to know how the subsystem will react to inputs at and beyond time t , we have to know what this value is at time t . We do not need to know anything more about the past inputs. Their effect on the future is entirely captured by the current value at t . The icon in Figure 2.2 includes i , an initial state value, which is needed to get things started.

An integrator operates in a time continuum. It integrates a continuous-time input signal, outputting at each time the cumulative area under the curve given by the input plus the initial state. Its state at any given time is that accumulated area plus the initial state. The counter actor in the previous section also has state, and that state is also an accumulation of past input values, but it operates

Probing Further: Discrete Signals

Discrete signals consist of a sequence of instantaneous events in time. Here, we make this intuitive concept precise.

Consider a signal of the form $e: \mathbb{R} \rightarrow \{absent\} \cup X$, where X is any set of values. This signal is a **discrete signal** if, intuitively, it is absent most of the time and we can count, in order, the times at which it is present (not absent). Each time it is present, we have a discrete event.

This ability to count the events in order is important. For example, if e is present at all rational numbers t , then we do not call this signal discrete. The times at which it is present cannot be counted in order. It is not, intuitively, a sequence of instantaneous events in time (it is a *set* of instantaneous events in time, but not a *sequence*).

To define this formally, let $T \subseteq \mathbb{R}$ be the set of times where e is present. Specifically,

$$T = \{t \in \mathbb{R} : e(t) \neq absent\}.$$

Then e is discrete if there exists a **one-to-one** function $f: T \rightarrow \mathbb{N}$ that is **order preserving**. Order preserving simply means that for all $t_1, t_2 \in T$ where $t_1 \leq t_2$, we have that $f(t_1) \leq f(t_2)$. The existence of such a one-to-one function ensures that we can count off the events *in temporal order*. Some properties of discrete signals are studied in Exercise 5.

(p. 155)

discretely.

For the integrator, the state $y(t)$ of the integrator at time t is a real number. Hence, we say that the **state space** of the integrator is $States = \mathbb{R}$. For the counter used in Figure 2.1, the state $s(t)$ at time t is a natural number, so $States \subset \mathbb{N}$. A practical parking garage has a finite number M of spaces, so the state space for the counter is

$$States = \{0, 1, 2, \dots, M\}.$$

The state space for the integrator is infinite (uncountably infinite, in fact). The state space for the garage counter is finite. Models with finite state spaces are called finite-state machines (FSMs). There are powerful analytical techniques available for such models, so we consider them next.

2.3 Finite-State Machines

- (p. 23) A **state machine** is a model of a system with **discrete dynamics** that at each **reaction** maps values of input signals to output values based on its current state. A **finite-state machine (FSM)** is a state machine where the set $States$ of possible states is finite.

If the number of states is reasonably small, then FSMs can be conveniently using a graphical notation like that in Figure 2.3. Here, each state is represented by a bubble, so for this diagram, the set of states is given by

$$States = \{\text{State1}, \text{State2}, \text{State3}\}.$$

At the beginning of each reaction, there is an **initial state**, **State1**, indicated in the diagram by a dangling arrow into it (and, for emphasis, a bold outline).

2.3.1 Transitions

Transitions between states govern the discrete dynamics of the state machine. A transition is represented as a curved arrow, as shown in Figure 2.3, going from one state to another. A transition

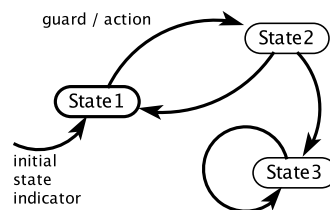


Figure 2.3: Visual notation for a finite state machine.

may also start and end at the same state, as illustrated with State3 in the figure. In this case, the transition is called a **self transition**.

In Figure 2.3, the transition from State1 to State2 is labeled with “guard / action.” The **guard** determines whether the transition may be taken on a reaction. The **action** specifies what outputs are produced on each reaction.

As in Section 2.1.2, we can denote the set of possible input values by *Inputs*. If the inputs are all pure signals, then we let $Inputs = 2^P$, where P is the set of input ports and 2^P is the powerset of P . A guard indicates which values of inputs trigger a transition, so $guard \subseteq Inputs$. For pure inputs, $guard \subseteq 2^P$. In this case, a guard can be given as a set of sets of ports. (p. 153)

Example 2.3: Figure 2.4 shows an FSM model for the garage counter. The set of states is $States = \{0, 1, 2, \dots, M\}$. The transition from state 0 to 1 has a guard written as $up \wedge \neg down$. This is a shorthand expression for the set of sets of ports

$$\{\{up\}\}.$$

This means that *up* is present and *down* is absent. If the guard expression had been simply *up*, then this would have been a shorthand for

$$\{\{up\}, \{up, down\}\},$$

which would have resulted in incorrectly counting cars when a car was arriving at the same time that another was departing.

When used with care, the notation in Figure 2.4 is unambiguous and complete. A guard is a **predicate** (a boolean-valued expression) that evaluates to *true* when the inputs match. In an expression like $up \wedge \neg down$, the subexpression *up* is a predicate that is true when the *up* input is present, and false otherwise. Similarly $\neg down$ is true when *down* is absent, and false otherwise. The symbol $\neg$ represents logical **negation**. The operator $\wedge$ is simply a logical **conjunction** (logical AND). You can also use the operator $\vee$, which is logical **disjunction** (logical OR).

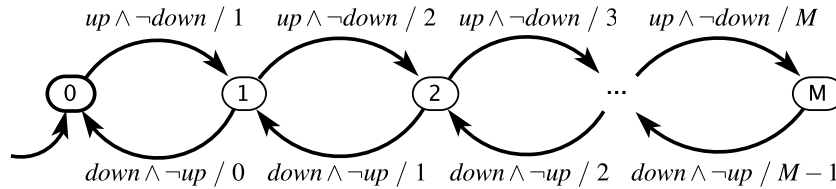


Figure 2.4: FSM model for the garage counter.

Inputs that are not pure signals are often amenable to a similar shorthand notation for the guards. In this case, technically, a guard specifies a subset of the possible input values that enable the transition. The subset can often be specified by a predicate that evaluates to true when the input values are in the subset.

Example 2.4: A major use of energy worldwide is in heating, ventilation, and air conditioning (HVAC) systems. Accurate models of temperature dynamics and temperature control systems can significantly improve energy conservation. Such modeling begins with a modest **thermostat**, which regulates temperature to maintain a **setpoint**, or target temperature. The word “thermostat” comes from Greek words for “hot” and “to make stand.”

Consider a thermostat that is a state machine with $States = \{\text{heating}, \text{cooling}\}$ as shown in Figure 2.5. Suppose the setpoint is 20 degrees Celsius. If the heater is on, then the thermostat allows the temperature to rise past the setpoint to 22 degrees. If the heater is off, then it allows the temperature to drop past the setpoint to 18 degrees. This strategy is called hysteresis (see box on page 30) and avoids **chattering**, where the heater would turn on and off rapidly when the temperature is close to the setpoint temperature.

There is a single input *temperature* and a single output *heat*. The output is a pure signal that is *present* when the transition from cooling to heating is taken and when the self loop on heating is taken. On all other reactions, the output is absent, as indicated by the empty expression after the slash on the other transitions.

The input *temperature* is not a pure signal. Assume that the temperature is given by a real number. Then

$$Inputs = (\{\text{absent}\} \cup \mathbb{R})^P$$

where $P = \{\text{temperature}\}$ is the set of input ports. That is, *Inputs* is the set of functions that map input ports (of which there is only one) into values (which are either *absent* or a real number).

The guards on each transition in Figure 2.5 specify subsets of *Inputs* in a compact way. For example, the guard $\text{temperature} > 18$ specifies the subset of functions where $Inputs(\text{temperature})$ is not absent and has a value greater than 18.

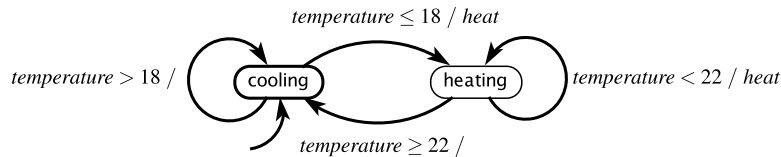


Figure 2.5: A model of a thermostat with hysteresis.

For an FSM, the set of possible output values is denoted by *Outputs*. An action on a transition, therefore, is simply a member of this set, $action \in Outputs$. It assigns a value to each output port.

Example 2.5: In Figure 2.4, the transitions specify output values after the slash. They do not specify the output port because there is only one output port, but we could optionally specify it using the notation like $count = 1$ for the transition from 0 to 1.

In Figure 2.5, the *heat* output is pure. Hence, after the slash on each transition we simply list *heat* to make it *present* and don't list it to make it *absent*.

Outputs may be denoted with shorthands similar to that of inputs. Suppose, for example, that an actor has two pure outputs named x and y . Then a transition could be labeled with “ x ” to denote that x is to be present and y is to be absent. Or it could be labeled with “ x,y ” to denote that both should be present. Or it could be empty (nothing after the slash) to indicate that all outputs are absent.

If an actor has multiple outputs that are not pure, then we can use a shorthand like $x = value_1, y = value_2$ to assign distinct values to the outputs. Any output ports not mentioned on a transition will be implicitly absent.

2.3.2 Implicit and Default Transitions

Suppose that an FSM is in a state and none of the outgoing transitions have guards that evaluate to true. For example, in Figure 2.4, if we are in state 0 and the input is $i = \{down\}$, then the guard on the outgoing transition is false. In this case, the state machine stays in state 0, and there is no output (the output is absent). Note that our previous informal description of the garage counter did not explicitly state what would happen in this circumstance. A major advantage of FSM models is that they are explicit about all possible behaviors. They are therefore amenable to formal checking, which determines whether the specified behaviors are in fact desirable behaviors. The informal specification cannot be subjected to such tests, or at least, not completely.

We can explicitly represent the implicit behavior when the input is $i = \{down\}$ using a **default transition**, shown in Figure 2.6. In that figure, the default transition is denoted with dashed lines and is labeled with “/” which indicates that there is no output. A default transition is taken whenever no other transition is enabled. If the default transition produces no output, then it need not be explicitly represented in the diagram. Sometimes, however, it is useful to have default transitions that do produce output.

Example 2.6: Figure 2.7 gives an example of an FSM that models the controller for a traffic light in a mostly-pedestrian zone. The traffic light turns green only when a sensor detects a vehicle at the intersection. The FSM has three states, red, yellow, and green,

corresponding to the three possible colors of the traffic light. There are four pure input signals to the controller: *isCar*, which models the presence of a car on the pavement, and *timeR*, *timeG*, and *timeY*, which model the expiration of timers monitoring time spent in the red, green, and yellow states respectively. Finally, there are three output signals: *sigR*, *sigG*, and *sigY* indicating the command to the device lighting the red, green, and yellow

Probing Further: Hysteresis

(p. 28)

The thermostat in Example 2.4 exhibits a particular form of state-dependent behavior called **hysteresis**. Hysteresis is used to prevent **chattering**. A system with hysteresis has the property that the absolute time scale is irrelevant. In Example 2.4, the input signal as a function of time is a signal of the form

$$temperature: \mathbb{R} \rightarrow \{absent\} \cup \mathbb{R}.$$

Hence, $temperature(t)$ is the temperature reading at time t , or *absent* if there is no temperature reading at that time. The output as a function of time has the form

$$heat: \mathbb{R} \rightarrow \{absent, present\}.$$

Suppose that instead of *temperature* the input is given by

$$temperature'(t) = temperature(\alpha \cdot t)$$

for a non-negative constant α . Then for this FSM, the output is $heat'$ given by

$$heat'(t) = heat(\alpha \cdot t).$$

Scaling the time axis at the input results in scaling the time axis at the output, so the absolute time scale is irrelevant.

An alternative implementation to for the thermostat would use a single temperature threshold, but instead would require that the heater remain on or off for at least a minimum amount of time, regardless of the temperature. The consequences of this design choice are explored in Exercise 2.

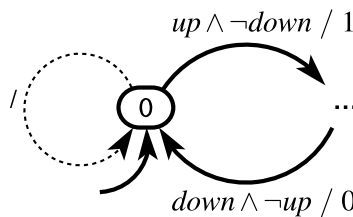


Figure 2.6: Explicit notation for a default transition.

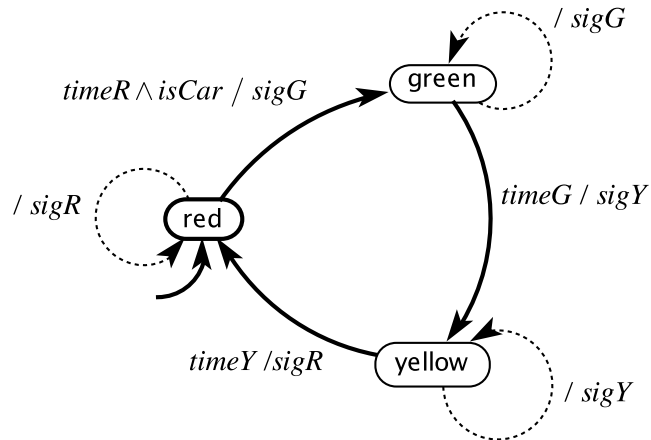


Figure 2.7: Finite-state machine modeling a traffic light controller.

lights at the intersection. Notice, in particular, how we model the set of input symbols causing the transition from `red` to `green`. Notice also the default transitions specifying what the FSM does in the default case in each state. Unlike the garage counter, the default transitions here do produce outputs.

2.3.3 Update Functions

The graphical notation we have described in fact gives a specific mathematical model of the dynamics of a state machine. Formally, a finite-state machine is a five-tuple

$$(\text{States}, \text{Inputs}, \text{Outputs}, \text{update}, \text{initialState})$$

Software Tools Supporting FSMs

FSMs have been used in software engineering for quite some time (Hopcroft and Ullman, 1979). A number of software tools support design and analysis of FSMs. Statecharts (Harel, 1987), a notation for concurrent composition of hierarchical FSMs, has influenced many of these tools. One of the first tools supporting the Statecharts notation is STATEMATE (Harel et al., 1990), which subsequently evolved into Rational Rose, now sold by IBM. Many variants of Statecharts have evolved (Beeck, 1994), and some variant is now supported by nearly every software engineering tool that provides UML (unified modeling language) capabilities (Booch et al., 1998). SyncCharts (André, 1996) is a particularly nice variant in that it borrows the rigorous semantics of Esterel (Berry and Gonthier, 1992) for composition of concurrent FSMs.

2.3. FINITE-STATE MACHINES

where

- *States* is a finite set of states;
- *Inputs* is a set of input values;
- *Outputs* is a set of output values;
- $update : States \times Inputs \rightarrow States \times Outputs$ is an **update function**, mapping a state and an input value to a *next* state and an output value;
- (p. 26) • *initialState* is the **initial state**.

We assume that inputs are discrete and that the FSM reacts in a sequence of reactions that we call **steps**. Thus, we can abstract away the passage of time and talk only about the sequence of reactions. Specifically, let $s : \mathbb{N} \rightarrow States$ be a function that gives the state of an FSM at step $n \in \mathbb{N}$. Initially, $s(0)initialState$.

Let $x : \mathbb{N} \rightarrow Inputs$ and $y : \mathbb{N} \rightarrow Outputs$ denote that input and output values at each step. Hence, $x(0) \in Inputs$ is the first input value and $y(0) \in Outputs$ is the first output value. The dynamics of the state machine are therefore given by the following equation:

$$(s(n+1), y(n)) = update(s(n), x(n)) \quad (2.3)$$

This gives the **next state** and output in terms of the **current state** and input. The *update* function encodes all the transitions, guards, and output specifications in an FSM. The term **transition function** is often used in place of update function.

Example 2.7: The FSM in Figure 2.7 can be formally represented as follows:

$$\begin{aligned} States &= \{\text{red}, \text{yellow}, \text{green}\} \\ Inputs &= 2^{\{\text{timeR}, \text{timeY}, \text{timeG}, \text{isCar}\}} \\ Outputs &= 2^{\{\text{sigR}, \text{sigY}, \text{sigG}\}} \\ initialState &= \text{red} \end{aligned}$$

Notice how the sets *Inputs* and *Outputs* are defined as power sets. For the update function, we give only a few representative mappings:

$$\begin{aligned} update(\text{red}, \{\text{timeR}, \text{isCar}\}) &= (\text{green}, \{\text{sigG}\}) \\ update(\text{red}, \{\text{timeR}\}) &= (\text{red}, \{\text{sigR}\}) \end{aligned}$$

The latter update corresponds to the default transition out of state **red**.

The state machines presented in this section have two important properties:

Determinacy: A state machine is said to be **deterministic** if, for each state, there is at most one transition enabled by each input value. The formal definition of an FSM given above ensures that it is deterministic, since *update* is a function, not a one-to-many mapping. The graphical notation with guards on the transitions, however, has no such constraint. Such a state machine will be deterministic only if the guards leaving each state are non-overlapping.

Receptiveness: A state machine is said to be **receptive** if, for each state, there is at least one transition possible on each input symbol. In other words, receptiveness ensures that a state machine is always ready to react to any input, and does not “get stuck” in any state. The formal definition of an FSM given above ensures that it is deterministic, since *update* is a function, not a [partial function](#). It is defined for every possible state and input value. Moreover, in our graphical notation, since we have implicit default transitions, we have ensured that all state machines specified in our graphical notation are also receptive. (p. 154)

It follows that if a state machine is both deterministic and receptive, for every state, there is *exactly* one transition possible on each input value.

2.3.4 Stuttering

In our definition of an FSM, the set *Inputs* represents the values that inputs can have when a reaction occurs. We have presumed that an FSM does not react if the inputs are all absent. We can equivalently presume, however, that the FSM does react when inputs are absent, but that when it reacts, it takes a default [self transition](#) that produces no outputs. Since this is a self transition, the state of the state machine does not change. When a state machine reacts by producing only absent outputs and not changing state, the reaction is called **stuttering**. No progress is made and nothing changes. Any number of stuttering reactions can be put between any two non-stuttering reactions without changing anything. (p. 27)

We can define state machines, however, that react to absent inputs and produce non-absent outputs. Such state machines are **spontaneous**. They are capable of producing outputs even when there are no inputs. Such state machines are typically nondeterministic. We study various forms of nondeterminism in state machines next.

2.4 Nondeterminism

Most interesting state machines react to inputs and produce outputs. These inputs must come from somewhere, and the outputs must go somewhere. We refer to this somewhere as the **environment** of the state machine.

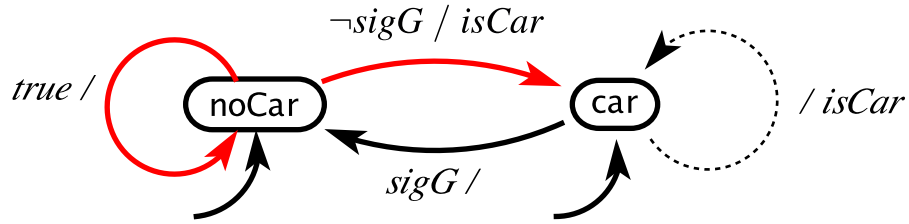


Figure 2.8: Nondeterministic FSM model of cars that arrive at a traffic light.

Example 2.8: The traffic light controller of Figure 2.7 has four pure input signals, one of which is called *isCar*. This signal has a *present* event when a car arrives at the intersection. The traffic light will remain red unless a car arrives. Some other subsystem is responsible for generating the *isCar* event. That other subsystem is part of the environment of the FSM in Figure 2.7.

A question becomes how to model the environment. In the traffic light example, we could construct a model of the city to serve this purpose, but this would likely be a very complicated model, and it is likely much more detailed than necessary. We want to ignore inessential details, and focus on the design of the traffic light. We can do this using a nondeterministic state machine.

Example 2.9: The FSM in Figure 2.8 models arrivals of cars at an intersection with a traffic light. This FSM has one input, *sigG*, and one output, *isCar*. Its initial state is *noCar*. The self-loop transition at this state has guard *true*, which is a shorthand for a guard that is always enabled (formally, any input values in *Inputs* will enable the transition). It will take the self loop to model no car arriving in that reaction. The transition to *car* models the arrival of a car and can occur if *sigG* is *absent*. Once in state *car*, it remains in that state until an input *sigG* arrives, at which time it will transition back to the *noCar* state.

In the example of Figure 2.8, the guards on the two transitions exiting state *noCar* overlap. The self loop is always enabled, and the other is enabled if *sigG* is *absent*. Thus, they can both be enabled in the same reaction. Such overlap is one of the marks of a nondeterministic FSM. On a color display, we show these transitions in red to emphasize that they are nondeterministic.

A second mark of a nondeterministic FSM is that it can produce outputs even when the input is absent (it is spontaneous). An absent input does not necessarily result in a stuttering reaction.

Specifically, in the example, if the FSM is in state `noCar`, then it may produce the output `isCar` even if the input is absent.

A third mark of a nondeterministic FSM is that it may have more than one initial state, indicated in a diagram by more than one state having the dangling arrow coming into it. In Figure 2.8, both states are marked as initial states, thus indicating that either one can serve as the initial state.

In all three cases, a nondeterministic FSM specifies a family of possible reactions rather than a single reaction. Operationally, all reactions in the family are possible. The nondeterministic FSM makes no statement at all about how *likely* the various reactions are. It is perfectly correct, for example, to always take the self loop in state `noCar` in Figure 2.8. A model that specifies likelihoods (in the form of probabilities) is a **stochastic model**, quite distinct from a nondeterministic model.

Formally, a **nondeterministic FSM** is also represented as a five-tuple

$$(States, Inputs, Outputs, possibleUpdates, initialStates)$$

where the first three elements are the same as for a deterministic FSM, but the last two are different:

- *States* is a finite set of states;
- *Inputs* is a set of input values;
- *Outputs* is a set of output values;
- $possibleUpdates : States \times Inputs \rightarrow 2^{States \times Outputs}$ is an **update relation**, mapping a state and an input value to a *set of possible* (next state, output value) pairs;
- *initialStates* is a set of initial states.

The form of the function *possibleUpdates* indicates there can be more than one next state and/or output value given a current state and input value. The codomain is the powerset of $States \times Outputs$. We refer to the *possibleUpdates* function as an *update relation*, to emphasize this difference. The term **transition relation** is also often used in place of update relation.

Note in addition that there can be more than one initial state for the nondeterministic FSM. This feature can be used to model uncertainty or hide unnecessary detail in the initialization of the finite-state system.

Example 2.10: The FSM in Figure 2.8 can be formally represented as follows:

$$\begin{aligned} States &= \{\text{noCar}, \text{car}\} \\ Inputs &= 2^{\{\text{sigG}\}} \\ Outputs &= 2^{\{\text{isCar}\}} \\ initialStates &= \{\text{noCar}, \text{Car}\} \end{aligned}$$

The update relation is given below:

$$\begin{aligned} \text{possibleUpdates}(\text{noCar}, \emptyset) &= \{(\text{noCar}, \emptyset), (\text{car}, \{\text{isCar}\})\} \\ \text{possibleUpdates}(\text{noCar}, \{\text{sigG}\}) &= \{(\text{noCar}, \emptyset)\} \\ \text{possibleUpdates}(\text{car}, \emptyset) &= \{(\text{car}, \{\text{isCar}\})\} \\ \text{possibleUpdates}(\text{car}, \{\text{sigG}\}) &= \{(\text{noCar}, \emptyset)\} \end{aligned}$$

Notice that only the first of these is actually nondeterministic. The other three each have only one possible outcome.

2.4.1 Uses of Non-Determinism

While nondeterminism is an interesting mathematical concept in itself, it has two major uses in modeling embedded systems:

Environment Modeling: It is often useful to hide irrelevant detail about how an environment operates, resulting in a non-deterministic FSM model. We have already seen one example of such environment modeling in Figure 2.8.

Specifications: System specifications impose requirements on some system features, while leaving other features unconstrained. Nondeterminism is a useful modeling technique in such settings as well. For example, consider a specification that the traffic light cycles through **red**, **green**, **yellow**, in that order, without regard for the timing between the outputs. The nondeterministic FSM in Figure 2.9 models this specification. The guard *true* on each transition indicates that the transition can be taken at any step. Technically, it means that each transitions is enabled for any input values in *Inputs*.

2.5 Behaviors and Traces

(p. 32) An FSM has discrete dynamics, reacting in a sequence of **steps**. As we did in Section 2.3.3, we can abstract away the passage of time and consider only the *sequence*. We do not need to talk explicitly about the amount of time that passes between steps, since this is actually irrelevant to the behavior of an FSM.

Leveraging this fact, we can model a deterministic FSM as a function F_d from sequences of input values to sequences of output values. This function has the form

$$F_d : (\mathbb{N} \rightarrow \text{Inputs}) \rightarrow (\mathbb{N} \rightarrow \text{Outputs})$$

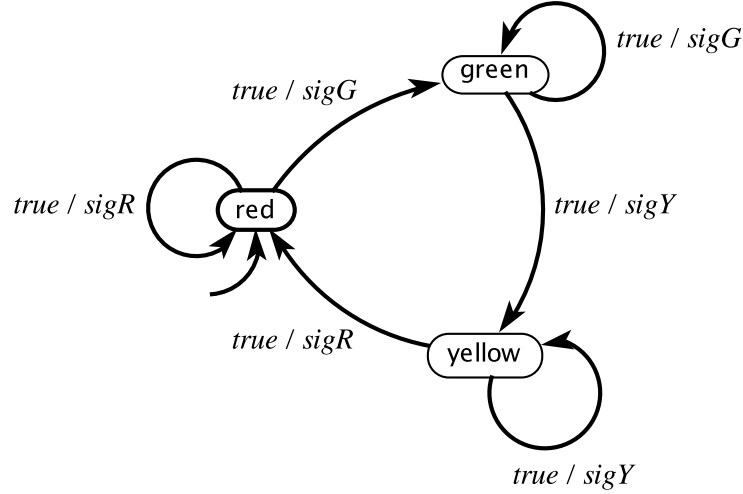


Figure 2.9: Nondeterministic FSM specifying order of signal lights, but not their timing.

A **behavior** of a deterministic FSM is a pair (x, y) such that $y = F_d(x)$, where each of x and y is a [sequence](#). (p. 157)

A non-deterministic FSM, on the other hand, can map a single input sequence to many possible output sequences, and hence is modeled by a function of form

$$F_n : (\mathbb{N} \rightarrow \text{Inputs}) \rightarrow 2^{(\mathbb{N} \rightarrow \text{Outputs})}$$

In this case, a behavior is a pair (x, y) such that $y \in F_n(x)$, but now y is a *set* of sequences. The set of all behaviors of a finite-state machine, deterministic or otherwise, is a subset of $(\mathbb{N} \rightarrow \text{Inputs}) \times (\mathbb{N} \rightarrow \text{Outputs})$.

It is often useful to be able to reason about the dynamics of system state. We can combine the notion of behavior with the dynamics of the state using the notion of a trace. For a deterministic FSM, a **trace** is a sequence

$$(x_0, s_0, y_0), (x_1, s_1, y_1), (x_2, s_2, y_2), \dots$$

such that $s_0 = \text{initialState}$, and for all $i \geq 0$, $(s_{i+1}, y_i) = \text{update}(s_i, x_i)$.

Example 2.11: Consider the deterministic FSM in Figure 2.7. Consider its reaction to the following input sequence:

$$\{\text{timeR}\}, \{\text{timeR}, \text{isCar}\}, \emptyset, \{\text{timeG}\}, \{\text{timeY}\}, \emptyset, \dots$$

The output sequence will be:

$$\{sigR\}, \{sigG\}, \{sigG\}, \{sigY\}, \{sigR\}, \{sigR\}, \dots$$

with corresponding trace:

$$\begin{aligned} &(\{timeR\}, red, \{sigR\}), \\ &(\{timeR, isCar\}, red, \{sigG\}), \\ &(\emptyset, green, \{sigG\}), \\ &(\{timeG\}, green, \{sigY\}), \\ &(\{timeY\}, yellow, \{sigR\}), \\ &(\emptyset, red, \{sigR\}), \\ &\dots \end{aligned}$$

For a non-deterministic FSM, the definition of a trace remains the same except that $s_0 \in initialStates$ and for all $i \geq 0$, $(s_{i+1}, y_i) \in possibleUpdates(s_i, x_i)$. Thus, notice that a non-deterministic FSM can have many possible behaviors and traces for the same input sequence.

Example 2.12: Consider the non-deterministic FSM in Figure 2.9. Consider the same input sequence as in Example 2.11:

$$\{timeR\}, \{timeR, isCar\}, \emptyset, \{timeG\}, \{timeY\}, \emptyset, \dots$$

One possible output sequence is

$$\{sigR\}, \{sigG\}, \{sigG\}, \{sigY\}, \{sigR\}, \{sigR\}, \dots$$

which is the same as the one in Example 2.11. However, the following output sequence is also possible with the very same input sequence:

$$\{sigR\}, \{sigR\}, \{sigR\}, \{sigG\}, \{sigG\}, \{sigG\}, \dots$$

In general, we can visualize the set of traces of a non-deterministic FSM reacting to a single input sequence as a **computation tree**, each of whose paths corresponds to a trace. Figure 2.10 shows the computation tree for the FSM of Figure 2.9 for the first three reactions with the above input sequence. Nodes in the tree are states and edges are labeled by the output values generated while transiting from state to state.

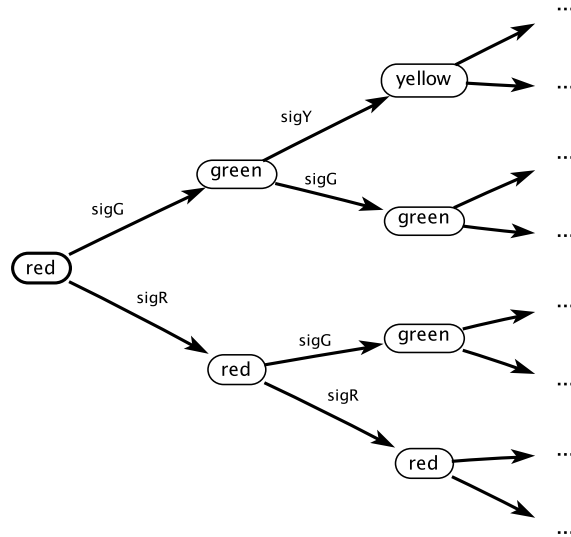


Figure 2.10: A computation tree for the FSM in Figure 2.9.

Exercises

1. Consider an event counter that is a simplified version of the counter in Section 2.1. It has an icon like this:



This actor starts with state i and upon arrival of an event at the input, increments the state and sends the new value to the output. Thus, e is a pure signal, and c has the form $c: \mathbb{R} \rightarrow \{absent\} \cup \mathbb{N}$, assuming $i \in \mathbb{N}$. Suppose you are to use such an event counter in a weather station to count the number of times that a temperature rises above some threshold. Your task in this exercise is to generate a reasonable input signal e for the event counter. You will create several versions. For all versions, you will design state machine whose input is a signal $\tau: \mathbb{R} \rightarrow \{absent\} \cup \mathbb{Z}$ that gives the current temperature (in degrees centigrade) once per hour. The output $e: \mathbb{R} \rightarrow \{absent, present\}$ will be a pure signal that goes to an event counter.

- (a) For the first version, your state machine should simply produce a *present* output whenever the input is *present* and greater than 38 degrees. Otherwise, the output should be *absent*.
- (b) For the second version, your state machine should have *hysteresis*. Specifically, it should produce a *present* output the first time the input is greater than 38 degrees, and subsequently, it should produce a *present* output anytime the input is greater than 38 degrees but has dropped below 36 degrees since the last time a *present* output was produced.

(p. 30)

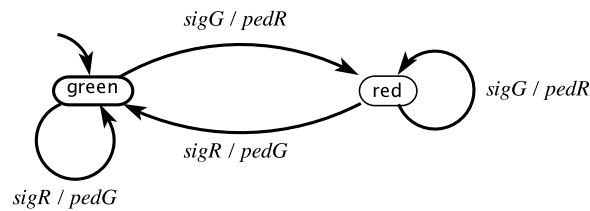


Figure 2.11: Finite-state machine modeling a pedestrian light controller.

- (c) For the third version, your state machine should implement the same hysteresis as in part (b), but also produce a *present* output at most once per day.
2. Consider a variant of the thermostat of example 2.4. In this variant, there is only one temperature threshold, and to avoid chattering the thermostat simply leaves the heat on or off for at least a fixed amount of time. In the initial state, if the temperature is less than or equal to 20 degrees Celsius, it turns the heater on, and leaves it on for at least 30 seconds. After that, if the temperature is greater than 20 degrees, it turns the heater off and leaves it off for at least 2 minutes. It turns it on again only if the temperature is less than or equal to 20 degrees.
- (a) Design an FSM that behaves as described, assuming it reacts exactly once every 30 seconds.
- (b) How many possible states does your thermostat have? Is this the smallest number of states possible?
- (p. 30) (c) Does this model thermostat have the [hysteresis](#) property?

3. Recall the traffic light controller of Figure 2.7. Consider connecting the outputs of this controller to a pedestrian light controller, whose FSM is given in Figure 2.11.

Construct a LabVIEW Statecharts model for the composition of the above two FSMs along with a deterministic FSM modeling the environment and generating input symbols *timeR*, *timeG*, *timeY*, and *isCar*. (For example, the environment FSM can use an internal counter to decide when to generate these symbols.) Use the LabVIEW “region” feature to construct the AND-state corresponding to the composition of these three FSMs.

Run the LabVIEW model and plot along the same time axis the signals *sigR*, *sigY*, *sigG*, *pedG*, *pedR* for one cycle of operation. Hand in printouts of your entire Statecharts model and the plots.

4. Consider the deterministic finite-state machine in Figure 2.12 that models a simple traffic light.
- (a) Formally write down the description of this FSM as a 5-tuple:

(*States, Inputs, Outputs, update, initialState*) .

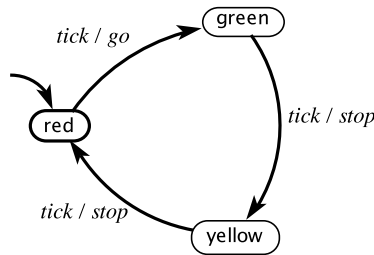


Figure 2.12: Deterministic finite-state machine for Exercise 4

- (b) Consider a trace of this FSM, including inputs, outputs, and states at each time point. Write down a prefix of this trace of length 4. How many traces can this FSM exhibit when the input has a *tick* at every reaction?
- (c) Now consider merging the **red** and **yellow** states into a single **stop** state. Transitions that pointed into or out of those states are now directed into or out of the new **stop** state. Other transitions and the inputs and outputs stay the same. The new **stop** state is the new initial state.
- Is the resulting state machine deterministic? Why or why not? If it is deterministic, give a prefix of the trace of length 4. If it is non-deterministic, draw the computation tree up to depth 4.
5. This exercise studies properties of discrete signals as formally defined in the sidebar on page 2.1.2. Specifically, we will show that discreteness is not a compositional property. That is, when combining two discrete behaviors in a single system, the resulting combination is not necessarily discrete.

- (a) Consider a **pure signal** x given by (p. 22)

$$x = \begin{cases} \text{present} & \text{if } x \text{ is a non-negative integer} \\ \text{absent} & \text{otherwise} \end{cases}$$

Show that this signal is discrete.

- (b) Consider a **pure signal** y given by (p. 22)

$$x = \begin{cases} \text{present} & \text{if } x = 1 - 1/n \text{ for any positive integer } n \\ \text{absent} & \text{otherwise} \end{cases}$$

Show that this signal is discrete.

- (c) Consider a signal w that is the merge of x and y in the previous two parts. That is, $w(t) = \text{present}$ if either $x(t) = \text{present}$ or $y(t) = \text{present}$, and is *absent* otherwise. Show that w is not discrete.

EXERCISES

- (d) Consider the example shown in Figure 2.1. Assume that each of the two signals *arrivals* and *departures* is discrete. Show that this does not imply that the output *count* is a discrete signal.

Chapter 3

Hybrid Systems¹

Contents

3.1	Modal Models	44
3.1.1	Actor Model for State Machines	44
3.1.2	Continuous Inputs	45
3.1.3	State Refinements	46
3.2	Classes of Hybrid Systems	47
3.2.1	Timed Automata	47
3.2.2	Higher-Order Dynamics	50
3.2.3	Supervisory control	55
3.3	Summary	60
	Exercises	62

Chapters 1 and 2 describe two very different modeling strategies, one focused on continuous dynamics and one on discrete dynamics. For continuous dynamics, we use [differential equations](#) and their corresponding [actor](#) models. For discrete dynamics, we use [state machines](#). (p. 4)
(p. 9)

Cyber-physical systems integrate physical dynamics and computational systems, so they commonly combine both discrete and continuous dynamics. In this chapter, we show that the modeling techniques of Chapters 1 and 2 can be combined, yielding what are known as **hybrid systems**. Hybrid system models are often much simpler and more understandable than brute-force models that constrain themselves to only one of the two styles in Chapters 1 and 2. They are a powerful tool for understanding real-world systems.

¹This chapter borrows heavily from [Lee and Varaiya \(2003\)](#).

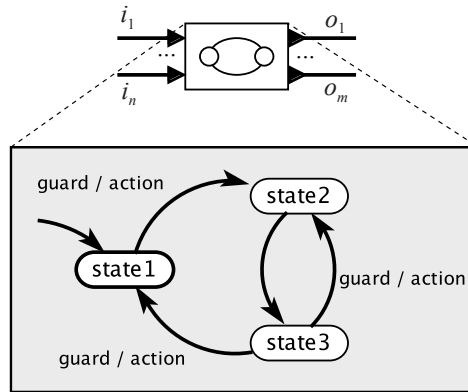


Figure 3.1: An FSM represented as an actor.

3.1 Modal Models

In this section, we show that state machines can be generalized to admit continuous inputs and outputs and to combine discrete and continuous dynamics.

3.1.1 Actor Model for State Machines

(p. 22) In Section 2.3.1 we explain that state machines have inputs defined by the set *Inputs* that may be pure signals or may carry a value. In either case, the state machine has a number of input ports, which in the case of pure signals are either present or absent, and in the case of valued signals have

(p. 23) a value at each reaction of the state machine.

(p. 26) We also explain in Section 2.3.1 that actions on transitions set the values of outputs. The outputs can also be represented by ports, and again the ports can carry pure signals or valued signals. In the case of pure signals, a transition that is taken specifies whether the output is present or absent, and in the case of valued signals, it assigns a value or asserts that the signal is absent. Outputs are presumed to be absent between transitions.

Given this input/output view of state machines, it is natural to think of a state machine as an actor, as illustrated in Figure 3.1. In that figure, we assume some number n of input ports named $i_1 \dots i_n$. At each reaction, these ports have a value that is either *present* or *absent* (if the port carries a pure signal) or a member of some set of values (if the port carries a valued signal). The outputs are similar. The guards on the transitions define subsets of possible values on input ports, and the actions assign values to output ports. Given such an actor model, it is straightforward to generalize

(p. 5) FSMs to admit continuous-time signals as inputs.

3.1.2 Continuous Inputs

We have so far assumed that state machines operate in a sequence of discrete [reactions](#). We have assumed that inputs and outputs are absent between reactions. We will now generalize this to allow inputs and outputs to be [continuous-time signals](#). (p. 23)
(p. 5)

In order to get state machine models to coexist with time-based models, we need to interpret state transitions to occur on the same time line used for the time-based portion of the system. The notion of discrete reactions described in Section 2.1 suffices for this purpose, but we will no longer require inputs and outputs to be absent between reactions. Instead, we will define a transition to occur at the earliest time that a guard on an outgoing transition from the current state becomes enabled. As before, during the time between reactions, a state machine is understood to be [stuttering](#). But the inputs and outputs are no longer required to be absent during that time. (p. 33)

Example 3.1: Consider a thermostat modeled as a state machine with states $\Sigma = \{\text{heating}, \text{cooling}\}$, shown in Figure 3.2. This is a variant of the model of Example 2.4 where instead of a discrete input that provides a temperature at each reaction, the input is a continuous-time signal $\tau: \mathbb{R} \rightarrow \mathbb{R}$ where $\tau(t)$ represents the temperature at time t . The initial state is **cooling**, and the transition out of this state is enabled at the earliest time t after the start time when $\tau(t) \leq 18$. In this example, we assume the outputs are pure signals *heatOn* and *heatOff*.

In the above example, the outputs are present only at the times the transitions are taken. We can also generalize FSMs to support continuous-time outputs, but to do this, we need the notion of state refinements.

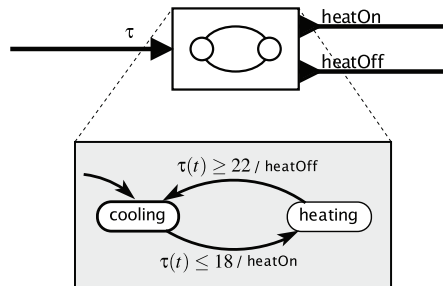


Figure 3.2: A thermostat modeled as an FSM with a continuous-time input signal.

3.1.3 State Refinements

A hybrid system associates with each state of an FSM a dynamic behavior. Our first (very simple) example uses this capability merely to produce continuous-time outputs.

Example 3.2: Suppose that instead of discrete outputs as in Example 3.1 we wish to produce a control signal whose value is 1 when the heat is on and 0 when the heat is off. Such a control signal could directly drive a heater. The thermostat in Figure 3.3 does this. In that figure, each state has a refinement that gives the value of the output h while the state machine is in that state.

In a hybrid system, the current state of the state machine has a **state refinement** that gives the dynamic behavior of the output as a function of the input. In the above simple example, the output is constant in each state, which is rather trivial dynamics, but hybrid systems can get much more elaborate.

The general structure of a hybrid system model is shown in Figure 3.4. In that figure, there is a two-state finite state machine. Each state is associated with a state refinement labeled in the figure as a “time-based system.” The state refinement defines dynamic behavior of the outputs and (possibly) additional continuous state variables. In addition, each transition can optionally specify **set actions**, which set the values of such additional state variables when a transition is taken. The example of Figure 3.3 is rather trivial, in that it has no continuous state variables, no output actions, and no set actions.

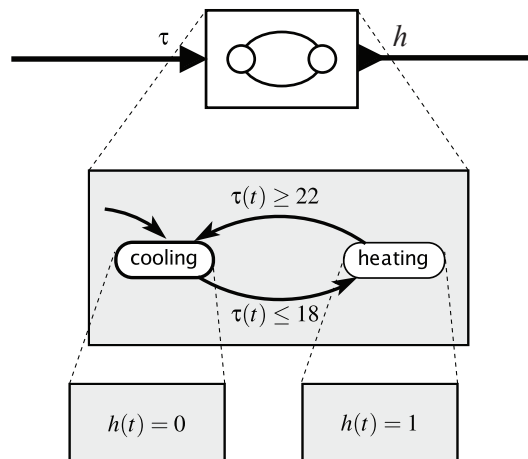


Figure 3.3: A thermostat with continuous-time output.

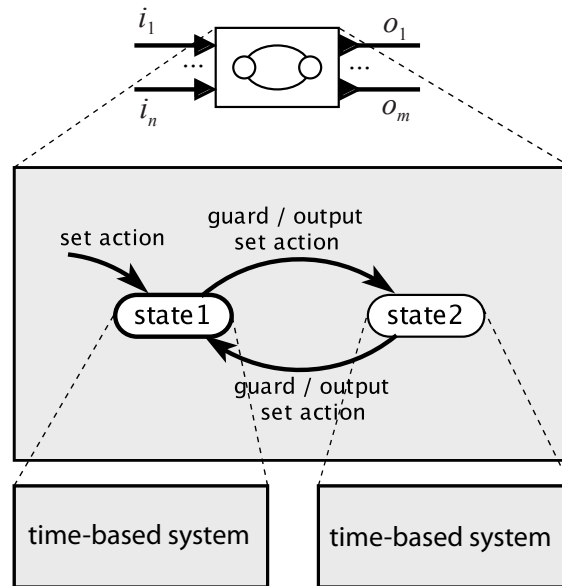


Figure 3.4: Notation for hybrid systems.

A hybrid system is sometimes called a **modal model** because it has a finite number of **modes**, one for each state of the FSM, and when it is in a mode, it has dynamics specified by the state refinement. The states of the FSM may be referred to as modes rather than states, which as we will see, helps prevent confusion with states of the refinements.

The next simplest such dynamics, besides the rather trivial constant outputs of Example 3.2 is found in timed automata, which we discuss next.

3.2 Classes of Hybrid Systems

Hybrid systems can be quite elaborate. In this section, we first describe a relatively simple form known as timed automata. We then illustrate more elaborate forms that model nontrivial physical dynamics and nontrivial control systems.

3.2.1 Timed Automata

Most cyber-physical systems require measuring the passage of time and performing actions at specific times. A device that measures the passage of time, a **clock**, has a particularly simple **dynamics**: (p. xi)

its state progresses linearly in time. In this section, we describe **timed automata**, which enable the construction of more complicated system from such simple clocks.

Timed automata are the simplest non-trivial hybrid systems. They are modal models where the time-based refinements have very simple dynamics; all they do is measure the passage of time. A clock is modeled by a first-order differential equation,

$$\forall t \in T_m, \quad \dot{s}(t) = a,$$

where $s: \mathbb{R} \rightarrow \mathbb{R}$ is a continuous-time signal, $s(t)$ is the value of the clock at time t , and $T_m \subset \mathbb{R}$ is the subset of time during which the hybrid system is in mode m . The rate of the clock, a , is a constant while the system is in this mode.

(p. 30)

Example 3.3: Recall the thermostat of Example 3.1, which uses **hysteresis** to prevent chattering. An alternative implementation that would also prevent chattering would use a single temperature threshold, but instead would require that the heater remain on or off for at least a minimum amount of time, regardless of the temperature. This design would not have the hysteresis property, but may be useful nonetheless. This can be modeled as a timed automaton as shown in Figure 3.5. In that figure, each state refinement has a clock, which is a continuous-time signal s with dynamics given by

$$\dot{s}(t) = 1.$$

The value $s(t)$ increases linearly with t . Note that in that figure, the state refinement is shown directly with the name of the state in the state bubble. This shorthand is convenient when the refinement is relatively simple.

(p. 46)

Notice that the initial state **cooling** has a **set action** on the dangling transition indicating the initial state, written as

$$s(t) := T_c.$$

(p. 27)

We use the notation “:=” to emphasize that this is an **assignment**, not a **predicate**. This action ensures that when the thermostat starts, it can immediately transition to the **heating** mode if the temperature $\tau(t)$ is less than or equal to 20 degrees. The other two transitions each have set actions that reset the clock s to zero. The portion of the guard that specifies $s(t) \geq T_h$ ensures that the heater will always be on for at least time T_h . The portion of the guard that specifies $s(t) \geq T_c$ specifies that once the heater goes off, it will remain off for at least time T_c .

(p. 28)

A possible execution of this timed automaton is shown in Figure 3.6. In that figure, we assume that the temperature is initially above the **setpoint** of 20 degrees, so the FSM remains in the **cooling** state until the temperature drops to 20 degrees. At that time t_1 , it can take the transition immediately because $s(t_1) > T_c$. The transition resets s to zero and turns on the heater. The heater will remain on until time $t_1 + T_h$, assuming that the

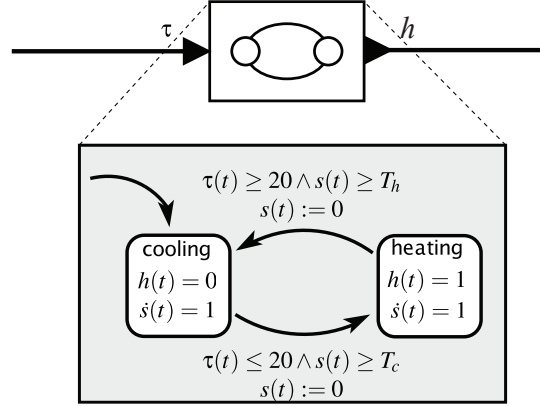


Figure 3.5: A timed automaton modeling a thermostat with a single temperature threshold and minimum times in each mode.

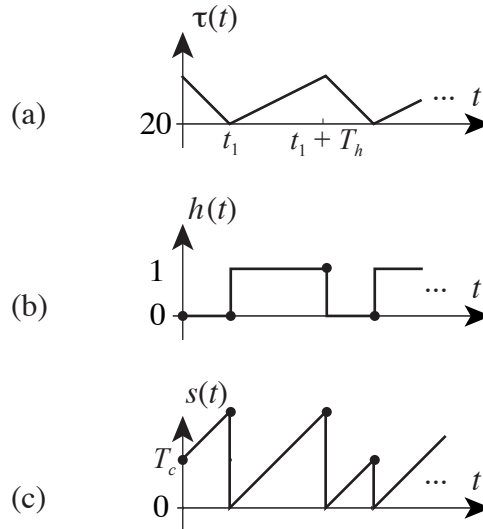


Figure 3.6: (a) A temperature input to the hybrid system of Figure 3.5, (b) the output h , and (c) the refinement state s .

temperature only rises when the heater is on. At time $t_1 + T_h$, it will transition back to the cooling state and turn the heater off. It will cool until at least time T_c elapses and until the temperature drops again to 20 degrees, at which point it will turn the heater back on.

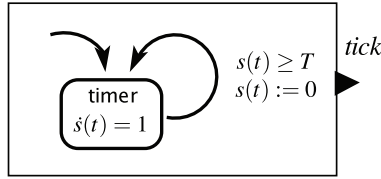


Figure 3.7: A timed automaton that generates a pure output event every T time units.

In the previous example the state of system at any time t is not only the mode, heating or cooling, but also the current value $s(t)$ of the clock. We call s a **continuous state** variable, whereas heating and cooling are **discrete states**. Thus, note that the term “state” for such a hybrid system can become confusing. The FSM has states, but so do the refinement systems (unless they are memory-less). When there is any possibility of confusion we explicitly refer to the states of the machine as modes.

Transitions between modes have actions associated with them. Sometimes, it is useful to have transitions from one mode back to itself, just so that the action can be realized. This is illustrated in the next example, which also shows a timed automaton that produces a pure output.

Example 3.4: The timed automaton in Figure 3.7 produces a pure output that will be present every T time units, starting at the time when the system begins executing. Notice that the guard on the transition, $s(t) \geq T$, is followed by an output action, *tick*, and a set action, $s(t) := 0$.

Figure 3.7 shows another notational shorthand that works well for simple diagrams. The automaton is shown directly inside the icon for its actor model.

3.2.2 Higher-Order Dynamics

In timed automata, all that happens in the time-based refinement systems is that time passes. Hybrid systems, however, are much more interesting when the behavior of the refinements is more complex.

Example 3.5: Consider the physical system depicted in Figure 3.8. Two sticky round masses are attached to springs. The springs are compressed or extended and then released. The masses oscillate on a frictionless table. If they collide, they stick together and oscillate together. After some time, the stickiness decays, and masses pull apart again.

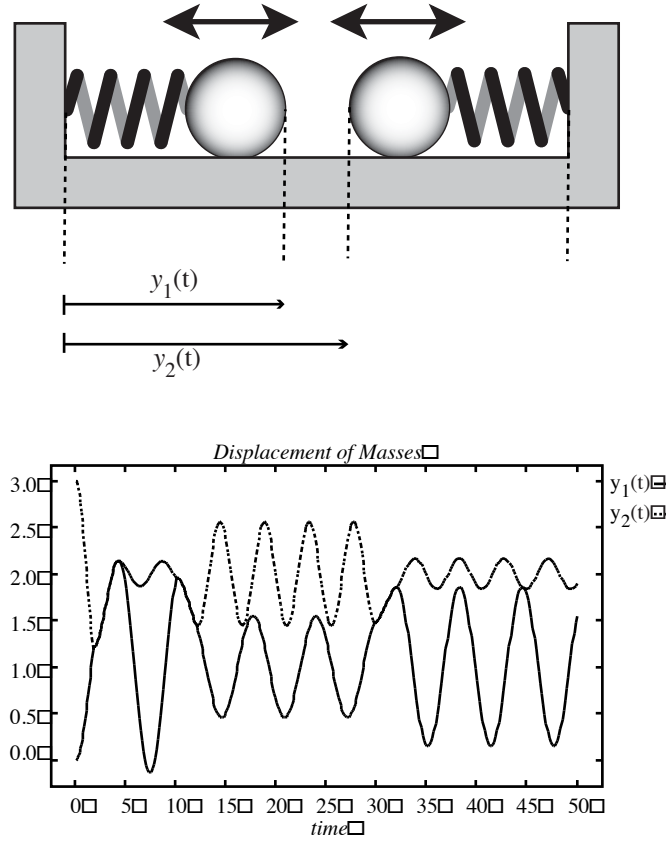


Figure 3.8: Sticky masses system considered in example 3.5.

A plot of the displacement of the two masses as a function of time is shown in the figure. Both springs begin compressed, so the masses begin moving towards one another. They almost immediately collide, and then oscillate together for a brief period until they pull apart. In this plot, they collide two more times, and almost collide a third time.

The physics of this problem is quite simple if we assume idealized springs. Let $y_1(t)$ denote the right edge of the left mass at time t , and $y_2(t)$ denote the left edge of the right mass at time t , as shown in Figure 3.8. Let p_1 and p_2 denote the neutral positions of the two masses, i.e. when the springs are neither extended nor compressed, so the force is zero. For an ideal spring, the force at time t on the mass is proportional to $p_1 - y_1(t)$ (for the left mass) and $p_2 - y_2(t)$ (for the right mass). The force is positive to the right and negative to the left.

Let the spring constants be k_1 and k_2 , respectively. Then the force on the left spring is $k_1(p_1 - y_1(t))$, and the force on the right spring is $k_2(p_2 - y_2(t))$. Let the masses be m_1 and m_2 respectively. Now we can use Newton's law, which relates force, mass, and

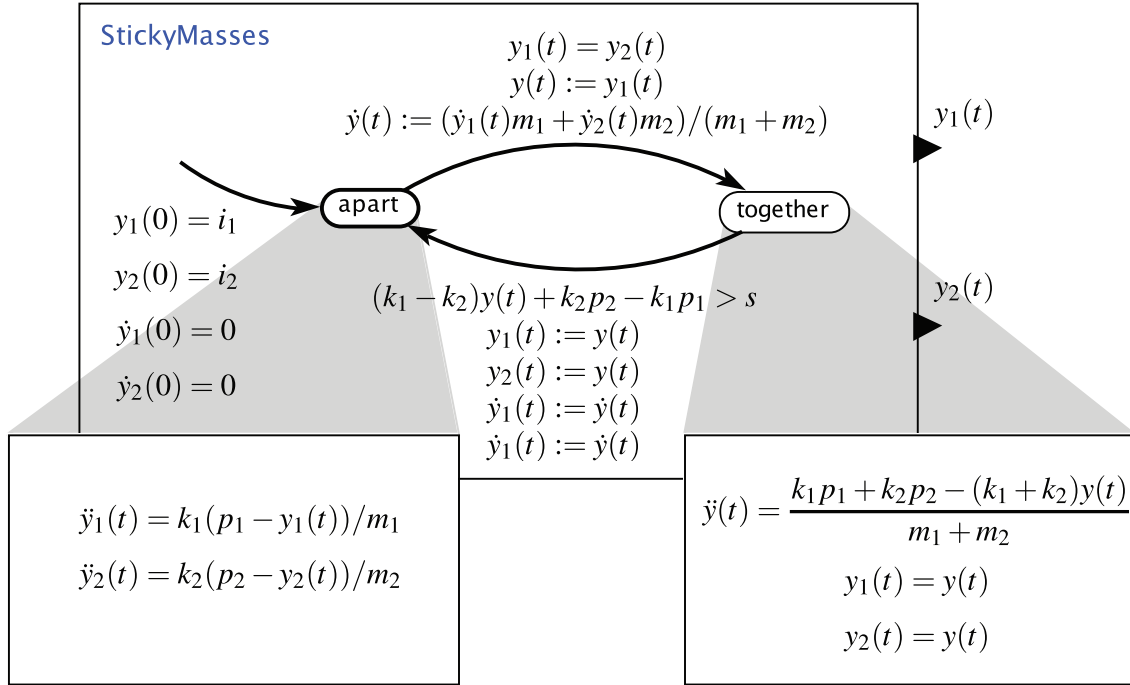


Figure 3.9: Hybrid system model for the sticky masses system considered in example 3.5.

acceleration,

$$f = ma.$$

The acceleration is the second derivative of the position with respect to time, which we write $\ddot{y}_1(t)$ and $\ddot{y}_2(t)$ respectively. Thus, as long as the masses are separate, their dynamics are given by

$$\ddot{y}_1(t) = k_1(p_1 - y_1(t))/m_1 \quad (3.1)$$

$$\ddot{y}_2(t) = k_2(p_2 - y_2(t))/m_2. \quad (3.2)$$

When the masses collide, however, the situation changes. With the masses stuck together, they behave as a single object with mass $m_1 + m_2$. This single object is pulled in opposite directions by two springs. While the masses are stuck together, $y_1(t) = y_2(t)$. Let

$$y(t) = y_1(t) = y_2(t).$$

The dynamics are then given by

$$\ddot{y}(t) = \frac{k_1 p_1 + k_2 p_2 - (k_1 + k_2)y(t)}{m_1 + m_2}. \quad (3.3)$$

It is easy to see now how to construct a hybrid systems model for this physical system. The model is shown in Figure 3.9. It has two modes, **apart** and **together**. The refinement of the **apart** mode is given by (3.1) and (3.2), while the refinement of the **together** mode is given by (3.3).

We still have work to do, however, to label the transitions. The initial transition is shown in Figure 3.9 entering the **apart** mode. Thus, we are assuming the masses begin apart. Moreover, this transition is labeled with a **set action** that sets the initial positions of the two masses to i_1 and i_2 and the initial velocities to zero. (p. 46)

The transition from **apart** to **together** has the guard

$$y_1(t) = y_2(t) .$$

This transition has a **set action** which assigns values to two **continuous state** variables $y(t)$ and $\dot{y}(t)$, which will represent the motion of the two masses stuck together. The value it assigns to $\dot{y}(t)$ conserves momentum. The momentum of the left mass is $\dot{y}_1(t)m_1$, the momentum of the right mass is $\dot{y}_2(t)m_2$, and the momentum of the combined masses is $\dot{y}(t)(m_1 + m_2)$. To make these equal, it sets (p. 50)

$$\dot{y}(t) = \frac{\dot{y}_1(t)m_1 + \dot{y}_2(t)m_2}{m_1 + m_2} .$$

The refinement of the **together** mode gives the dynamics of y and simply sets $y_1(t) = y_2(t) = y(t)$, since the masses are moving together. The transition from **apart** to **together** sets $y(t)$ equal to $y_1(t)$ (it could equally well have chosen $y_2(t)$, since these are equal).

The transition from **together** to **apart** has the more complicated guard

$$(k_1 - k_2)y(t) + k_2p_2 - k_1p_1 > s ,$$

where s represents the stickiness of the two masses. This guard is satisfied when the right-pulling force on the right mass exceeds the right-pulling force on the left mass by more than the stickiness. The right-pulling force on the right mass is simply

$$f_2(t) = k_2(p_2 - y(t))$$

and the right-pulling force on the left mass is

$$f_1(t) = k_1(p_1 - y(t)) .$$

Thus,

$$f_2(t) - f_1(t) = (k_1 - k_2)y(t) + k_2p_2 - k_1p_1 .$$

When this exceeds the stickiness K , then the masses pull apart.

An interesting elaboration on this example, considered in problem 8, modifies the together mode so that the stickiness is initialized to a starting value, but then decays according to the differential equation

$$\dot{s}(t) = -as(t)$$

where $s(t)$ is the stickiness at time t , and a is some positive constant. In fact, it is the dynamics of such an elaboration that is plotted in Figure 3.8.

As in Example 3.4, it is sometimes useful to have hybrid system models with only one state. The actions on one or more state transitions define the discrete event behavior that combines with the time-based behavior.

Example 3.6: Consider a bouncing ball. At time $t = 0$, the ball is dropped from a height $y(0) = h_0$, where h_0 is the initial height in meters. It falls freely. At some later time t_1 it hits the ground with a velocity $\dot{y}(t_1) < 0$ m/s (meters per second). A *bump* event is produced when the ball hits the ground. The collision is **inelastic** (meaning that kinetic energy is lost), and the ball bounces back up with velocity $-a\dot{y}(t_1)$, where a is constant with $0 < a < 1$. The ball will then rise to a certain height and fall back to the ground repeatedly.

The behavior of the bouncing ball can be described by the hybrid system of Figure 3.10. There is only one mode, called **free**. When it is not in contact with the ground, we know that the ball follows the second-order differential equation,

$$\ddot{y}(t) = -g, \tag{3.4}$$

where $g = 9.81$ m/sec² is the acceleration imposed by gravity. The continuous state variables of the **free** mode are

$$s(t) = \begin{bmatrix} y(t) \\ \dot{y}(t) \end{bmatrix}$$

with the initial conditions $y(0) = h_0$ and $\dot{y}(0) = 0$. It is then a simple matter to rewrite (3.4) as a first-order differential equation,

$$\dot{s}(t) = f(s(t)) \tag{3.5}$$

for a suitably chosen function f .

At the time t_1 when the ball first hits the ground, the guard

$$y(t) = 0$$

(p. 50)

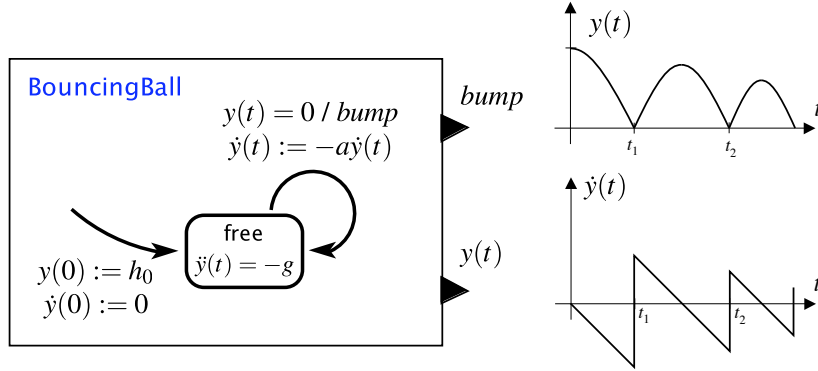


Figure 3.10: The motion of a bouncing ball may be described as a hybrid system with only one mode. The system outputs a *bump* each time the ball hits the ground, and also outputs the position of the ball. The position and velocity are plotted versus time at the right.

is satisfied, and the self-loop transition is taken. The output *bump* is produced, and the set action $\dot{y}(t) := -a\dot{y}(t)$ changes $\dot{y}(t_1)$ to have value $-a\dot{y}(t_1)$. Then (3.4) is followed again until the guard becomes true again. (p. 46)

By integrating (3.4) we get, for all $t \in (0, t_1)$,

$$\begin{aligned}\dot{y}(t) &= -gt, \\ y(t) &= y(0) + \int_0^t \dot{y}(\tau) d\tau = h_0 - \frac{1}{2}gt^2.\end{aligned}$$

So $t_1 > 0$ is determined by $y(t_1) = 0$. It is the solution to the equation

$$h_0 - \frac{1}{2}gt^2 = 0.$$

Thus,

$$t_1 = \sqrt{2h_0/g}.$$

Figure 3.10 plots the continuous state versus time.

3.2.3 Supervisory control

A control system involves four components. There is a system called the **plant**—the physical process that is to be controlled; the environment in which the plant operates; the sensors that measure some variables of the plant and the environment; and the controller that determines the mode tran-

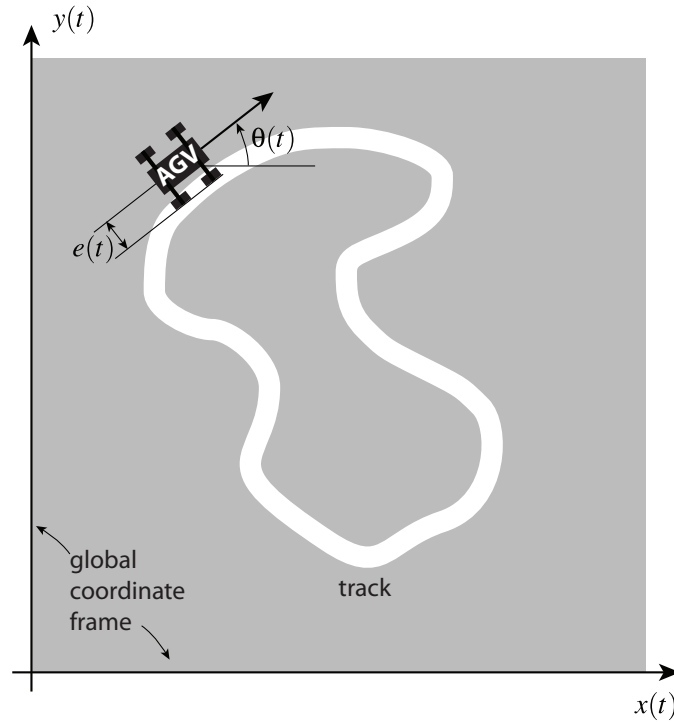


Figure 3.11: Illustration of the automated guided vehicle of example 3.7. The vehicle is following a curved painted track, and has deviated from the track by a distance $e(t)$. The coordinates of the vehicle at time t with respect to the global coordinate frame are $(x(t), y(t), \theta(t))$.

sition structure and selects the time-based inputs to the plant. The controller has two levels: the **supervisory control** that determines the mode transition structure, and the **low-level control** that determines the time-based inputs to the plant. Intuitively, the supervisory controller determines which of several strategies should be followed, and the low-level controller implements the selected strategy. Hybrid systems are ideal for modeling such two-level controllers. We show how through a detailed example.

Example 3.7: Consider an **automated guided vehicle (AGV)** that moves along a closed track painted on a warehouse or factory floor. We will design a controller so that the vehicle closely follows the track.

The vehicle has two degrees of freedom. At any time t , it can move forward along its body axis with speed $u(t)$ with the restriction that $0 \leq u(t) \leq 10$ mph (miles per hour). It can also rotate about its center of gravity with an angular speed $\omega(t)$ restricted to $-\pi \leq$

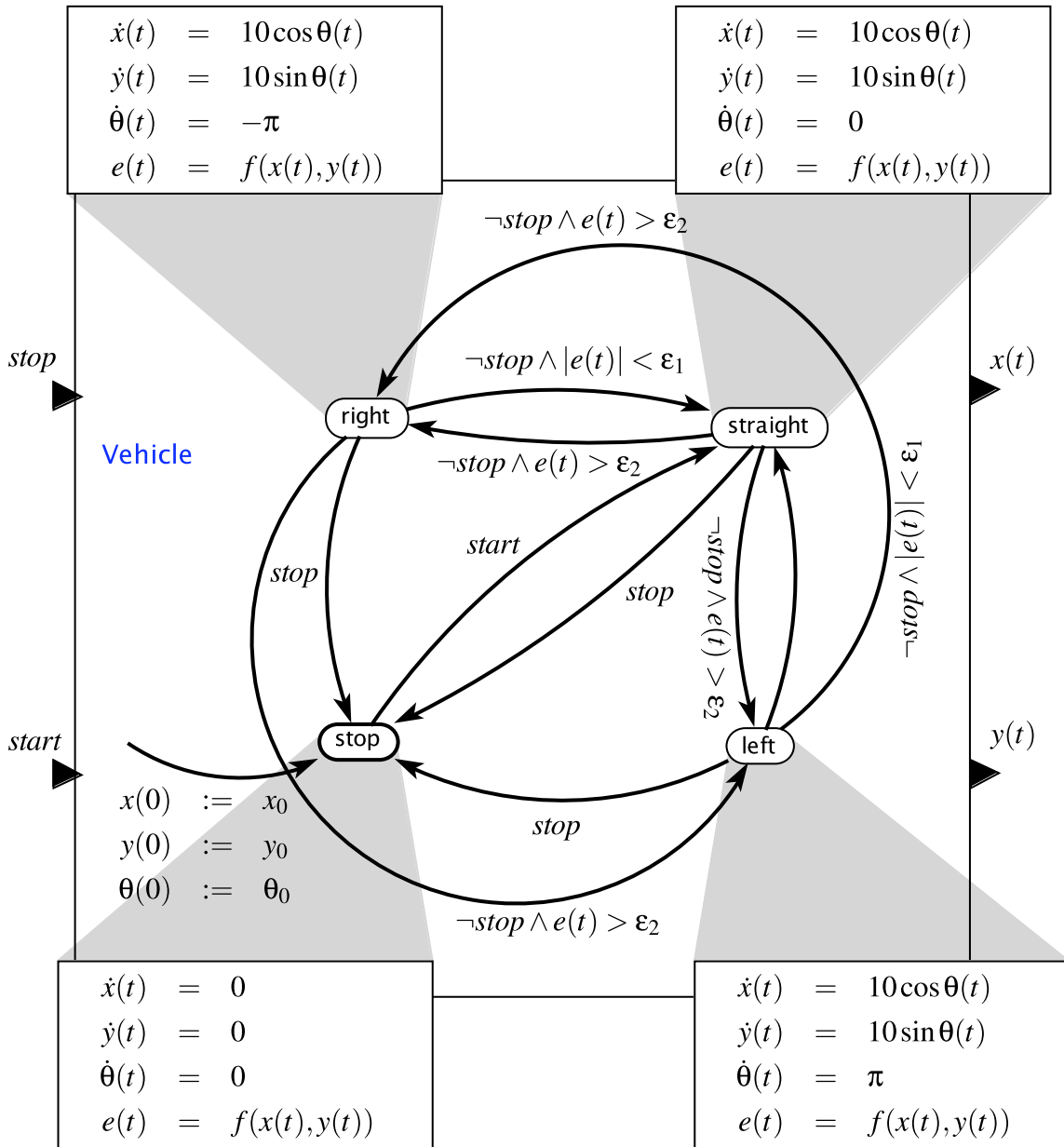


Figure 3.12: The automatic guided vehicle of example 3.7 has four modes: stop, straight, left, right.

$\omega(t) \leq \pi$ radians/second. We ignore the inertia of the vehicle, so we assume that we can instantaneously change the velocity or angular speed.

Let $(x(t), y(t)) \in \mathbb{R}^2$ be the position relative to some fixed coordinate frame and $\theta(t) \in (-\pi, \pi]$ be the angle (in radians) of the vehicle at time t , as shown in Figure 3.11. In terms of this coordinate frame, the motion of the vehicle is given by a system of three differential equations,

$$\begin{aligned}\dot{x}(t) &= u(t) \cos \theta(t), \\ \dot{y}(t) &= u(t) \sin \theta(t), \\ \dot{\theta}(t) &= \omega(t).\end{aligned}\tag{3.6}$$

Equations (3.6) describe the plant. The environment is the closed painted track. It could be described by an equation. We will describe it indirectly below by means of a sensor.

The two-level controller design is based on a simple idea. The vehicle always moves at its maximum speed of 10 mph. If the vehicle strays too far to the left of the track, the controller steers it towards the right; if it strays too far to the right of the track, the controller steers it towards the left. If the vehicle is close to the track, the controller maintains the vehicle in a straight direction. Thus the controller guides the vehicle in four modes, left, right, straight, and stop. In stop mode, the vehicle comes to a halt.

The following differential equations govern the AGV's motion in the refinements of the four modes. They describe the low-level controller, i.e. the selection of the time-based plant inputs in each mode.

straight

$$\begin{aligned}\dot{x}(t) &= 10 \cos \theta(t) \\ \dot{y}(t) &= 10 \sin \theta(t) \\ \dot{\theta}(t) &= 0\end{aligned}$$

left

$$\begin{aligned}\dot{x}(t) &= 10 \cos \theta(t) \\ \dot{y}(t) &= 10 \sin \theta(t) \\ \dot{\theta}(t) &= \pi\end{aligned}$$

right

$$\begin{aligned}\dot{x}(t) &= 10 \cos \theta(t) \\ \dot{y}(t) &= 10 \sin \theta(t) \\ \dot{\theta}(t) &= -\pi\end{aligned}$$

stop

$$\begin{aligned}\dot{x}(t) &= 0 \\ \dot{y}(t) &= 0 \\ \dot{\theta}(t) &= 0\end{aligned}$$

In the **stop** mode, the vehicle is stopped, so $x(t)$, $y(t)$, and $\theta(t)$ are constant. In the **left** mode, $\theta(t)$ increases at the rate of π radians/second, so from Figure 3.11 we see that the vehicle moves to the left. In the **right** mode, it moves to the right. In the **straight** mode, $\theta(t)$ is constant, and the vehicle moves straight ahead with a constant heading. The refinements of the four modes are shown in the boxes of Figure 3.12.

We design the supervisory control governing transitions between modes in such a way that the vehicle closely follows the track, using a sensor that determines how far the vehicle is to the left or right of the track. We can build such a sensor using photodiodes. Let's suppose the track is painted with a light-reflecting color, whereas the floor is relatively dark. Underneath the AGV we place an array of photodiodes as shown in Figure 3.13. The array is perpendicular to the AGV body axis. As the AGV passes over the track, the diode directly above the track generates more current than the other diodes. By comparing the magnitudes of the currents through the different diodes, the sensor estimates the displacement $e(t)$ of the center of the array (hence, the center of the AGV) from the track. We adopt the convention that $e(t) < 0$ means that the AGV is to the right of the track and $e(t) > 0$ means it is to the left. We model the sensor output as a function f of the AGV's position,

$$\forall t, \quad e(t) = f(x(t), y(t)).$$

The function f of course depends on the environment—the track. We now specify the supervisory controller precisely. We select two thresholds, $0 < \epsilon_1 < \epsilon_2$, as shown in Figure 3.13. If the magnitude of the displacement is small, $|e(t)| < \epsilon_1$, we consider that the AGV is close enough to the track, and the AGV can move straight ahead, in **straight** mode. If $e(t) > \epsilon_2$ ($e(t)$ is large and positive), the AGV has strayed too far to the left and must be steered to the right, by switching to **right** mode. If $e(t) < -\epsilon_2$ ($e(t)$ is large and negative), the AGV has strayed too far to the right and must be steered to the left, by switching to **left** mode. This control logic is captured in the mode transitions of Figure 3.12. The inputs are pure signals *stop* and *start*. These model an operator that can stop or start the AGV. There is no continuous-time input. The outputs represent the position of the vehicle, $x(t)$ and $y(t)$. The initial mode is **stop**, and the initial values of its refinement are (x_0, y_0, θ_0) .

We analyze how the AGV will move. Figure 3.14 sketches one possible trajectory. Initially the vehicle is within distance ϵ_1 of the track, so it moves straight. At some later time, the vehicle goes too far to the left, so the guard

$$\neg \text{stop} \wedge e(t) > \epsilon_2$$

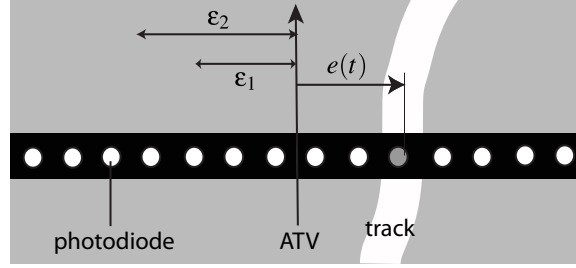


Figure 3.13: An array of photodiodes under the AGV is used to estimate the displacement e of the AGV relative to the track. The photodiode directly above the track generates more current.

is satisfied, and there is a mode switch to **right**. After some time, the vehicle will again be close enough to the track, so the guard

$$\neg stop \wedge |e(t)| < \epsilon_1$$

is satisfied, and there is a mode switch to **straight**. Some time later, the vehicle is too far to the right, so the guard

$$\neg stop \wedge e(t) < -\epsilon_2$$

is satisfied, and there is a mode switch to **left**. And so on.

The example illustrates the four components of a control system. The plant is described by the differential equations (3.6) that govern the evolution of the continuous state at time t , $(x(t), y(t), \theta(t))$, in terms of the plant inputs u and ω . The second component is the environment—the closed track. The third component is the sensor, whose output at time t , $e(t) = f(x(t), y(t))$, gives the position of the AGV relative to the track. The fourth component is the two-level controller. The supervisory controller comprises the four modes and the guards that determine when to switch between modes. The low-level controller specifies how the time-based inputs to the plant, u and ω , are selected in each mode.

3.3 Summary

Hybrid systems provide a bridge between time-based models and state-machine models. The combination of the two families of models provides a rich framework for describing real-world systems. There are two key ideas. First, discrete events are embedded in a time base. Second, a hierarchical description is particularly useful, where the system undergoes discrete transitions between different modes of operation. Associated with each mode of operation is a time-based system called the

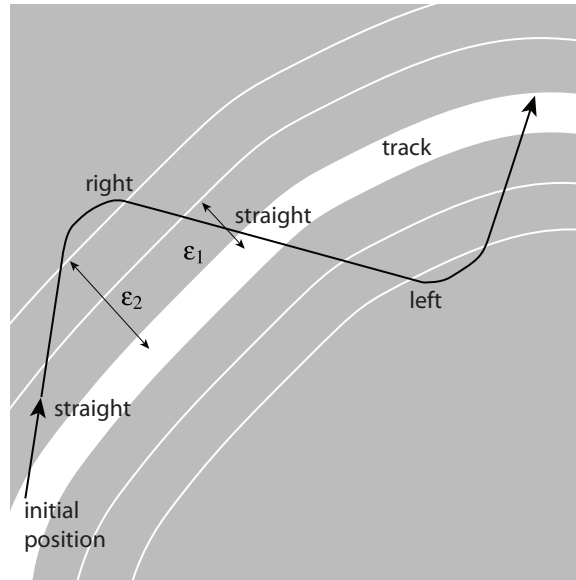


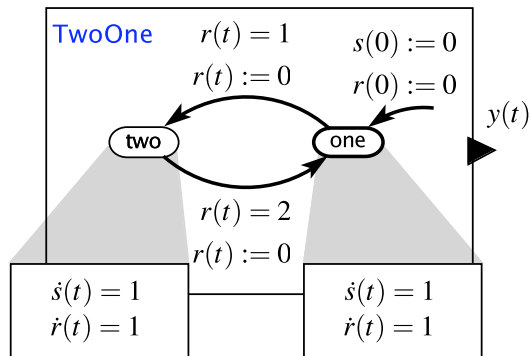
Figure 3.14: A trajectory of the AGV, annotated with modes.

refinement of the mode. Mode transitions are taken when guards that specify the combination of inputs and continuous states are satisfied. The action associated with a transition, in turn, sets the continuous state in the destination mode.

The behavior of a hybrid system is understood using the tools of state machine analysis for mode transitions the tools of time-based analysis for the refinement systems. The design of hybrid systems similarly proceeds on two levels: state machines are designed to achieve the appropriate logic of mode transitions, and refinement systems are designed to secure the desired time-based behavior in each mode.

Exercises

1. Construct (on paper is sufficient) a timed automaton similar to that of Figure 3.7 which produces *tick* at times 1, 2, 3, 5, 6, 7, 8, 10, 11, $\dots$. That is, ticks are produced with intervals between them of 1 second (three times) and 2 seconds (once).
2. The objective of this problem is to understand a timed automaton, and then to modify it as specified.
 - (a) For the timed automaton shown below, describe the output y . Avoid imprecise or sloppy notation.



- (b) Assume there is a new pure input *reset*, and that when this input is present, the hybrid system starts over, behaving as if it were starting at time 0 again. Modify the hybrid system from part (a) to do this.
3. You have an analog source that produces a pure tone. You can switch the source on or off by the input event *on* or *off*. Construct a timed automaton that provides the *on* and *off* signals as outputs, to be connected to the inputs of the tone generator. Your system should behave as follows. Upon receiving an input event *ring*, it should produce an 80 ms-long sound consisting of three 20 ms-long bursts of the pure tone separated by two 10 ms intervals of silence. What does your system do if it receives two *ring* events that are 50 ms apart?
4. Automobiles today have the features listed below. Implement each feature as a timed automaton.
 - (a) The dome light is turned on as soon as any door is opened. It stays on for 30 seconds after all doors are shut. What sensors are needed?
 - (b) Once the engine is started, a beeper is sounded and a red light warning is indicated if there are passengers that have not buckled their seat belt. The beeper stops sounding after 30 seconds, or as soon the seat belts are buckled, whichever is sooner. The warning light is on all the time the seat belt is unbuckled. **Hint:** Assume the sensors provide a *warn* event when the ignition is turned on and there is a seat with passenger not buckled

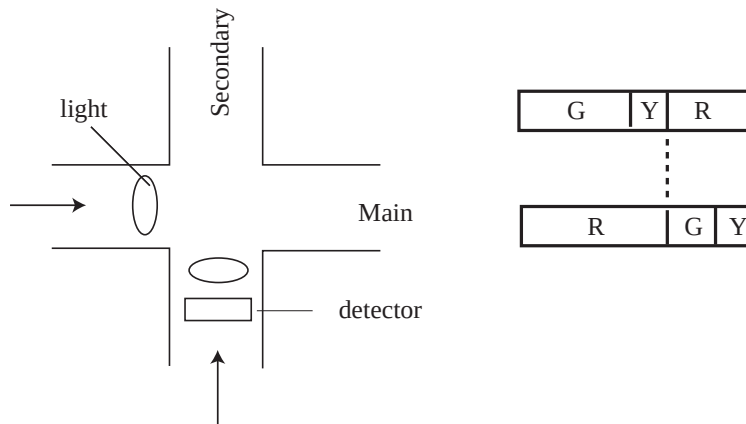


Figure 3.15: Traffic lights control the intersection of a main street and a secondary street. A detector senses when a vehicle crosses it. The red phase of one light must coincide with the green and yellow phases of the other light.

in, or if the ignition is already on and a passenger sits in a seat without buckling the seatbelt. Assume further that the sensors provide a *noWarn* event when a passenger departs from a seat, or when the buckle is buckled, or when the ignition is turned off.

5. A programmable thermostat allows you to select 4 times, $0 \leq T_1 \leq \dots \leq T_4 < 24$ (for a 24-hour cycle) and the corresponding [setpoint](#) temperatures $a_1, \dots, a_4$. Construct a timed automaton that sends the event a_i to the heating systems controller. The controller maintains the temperature close to the value a_i until it receives the next event. How many timers and modes do you need? (p. 28)
6. Figure 3.15 depicts the intersection of two one-way streets, called Main and Secondary. A light on each street controls its traffic. Each light goes through a cycle consisting of a red (R), green (G), and yellow (Y) phases. It is a safety requirement that when one light is in its green or yellow phase, the other is in its red phase. The yellow phase is always 5 seconds long.

The traffic lights operate as follows. A sensor in the secondary road detects a vehicle. While no vehicle is detected, there is a 4 minute-long cycle with the main light having 3 minutes of green, 5 seconds of yellow, and 55 seconds of red. The secondary light is red for 3 minutes and 5 seconds (while the main light is green and yellow), green for 50 seconds, then yellow for 5 seconds.

If a vehicle is detected on the secondary road, the traffic light quickly gives a right of way to the secondary road. When this happens, the main light aborts its green phase and immediately switches to its 5 second yellow phase. If the vehicle is detected while the main light is yellow or red, the system continues as if there were no vehicle.

Design a hybrid system that controls the lights. Let this hybrid system have six pure outputs, one for each light, named mG , mY , and mR , to designate the main light being green, yellow, or red, respectively, and sG , sY , and sR , to designate the secondary light being green, yellow, or red, respectively. These signals should be generated to turn on a light. You can implicitly assume that when one light is turned on, whichever has been on is turned off.

7. For the bouncing ball of example 3.6 let t_n be the time when the ball hits the ground for the n -th time, and let $v(n) = \dot{y}(t_n)$ be the velocity at that time.
 - (a) Find a relation between $v(n+1)$ and $v(n)$ and then calculate $v(n)$ in terms of $v(1)$.
 - (b) Obtain t_n in terms of $v(n)$.
 - (c) Calculate the maximum height reached by the ball after successive bumps.

8. Elaborate the hybrid system model of figure 3.9 so that in the *together* mode, the stickiness decays according to the differential equation

$$\dot{s}(t) = -as(t)$$

where $s(t)$ is the stickiness at time t , and a is some positive constant. On the transition into this mode, the stickiness should be initialized to some starting stickiness b .

9. Show that the trajectory of the AGV of figure 3.12 while it is in *left* or *right* mode is a circle. What is the radius of this circle, and how long does it take to complete a circle?
10. Consider Figure 3.16 depicting a system comprising two tanks containing water. Each tank is leaking at a constant rate. Water is added at a constant rate to the system through a hose, which at any point in time is filling either one tank or the other. It is assumed that the hose can switch between the tanks instantaneously. For $i \in \{1, 2\}$, let x_i denote the volume of water in Tank i and $v_i > 0$ denote the constant flow of water out of Tank i . Let w denote the constant flow of water into the system. The objective is to keep the water volumes above r_1 and r_2 , respectively, assuming that the water volumes are above r_1 and r_2 initially. This is to be achieved by a controller that switches the inflow to Tank 1 whenever $x_1(t) \leq r_1(t)$ and to Tank 2 whenever $x_2(t) \leq r_2(t)$.

The hybrid automaton representing this two-tank system is given in Figure 3.17.

Answer the following questions:

- (a) Construct a model of this hybrid automaton in Ptolemy II, LabVIEW, or Simulink. Use the following parameter values: $r_1 = r_2 = 0$, $v_1 = v_2 = 0.5$, and $w = 0.75$. Set the initial state to be $(q_1, (0, 1))$. (That is, initial value $x_1(0)$ is 0 and $x_2(0)$ is 1.)
Verify that this hybrid automaton is Zeno. What is the reason for this Zeno behavior? Simulate your model and plot how x_1 and x_2 vary as a function of time t , simulating long enough to illustrate the Zeno behavior.

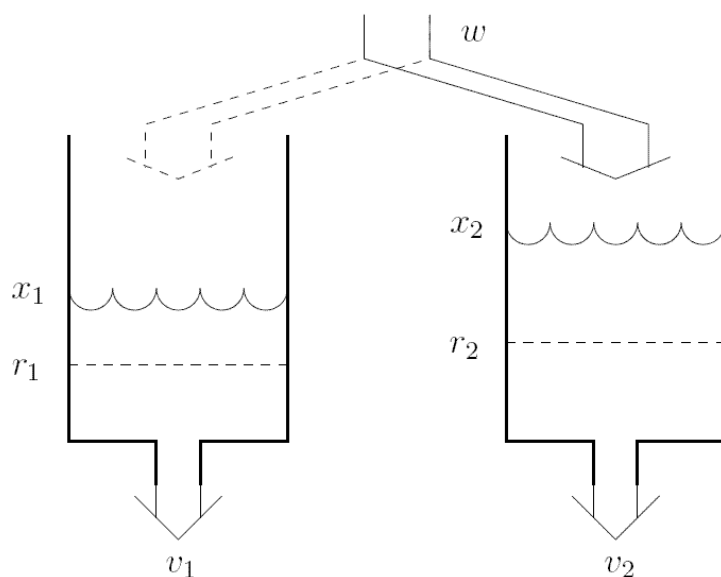


Figure 3.16: Water tank system.

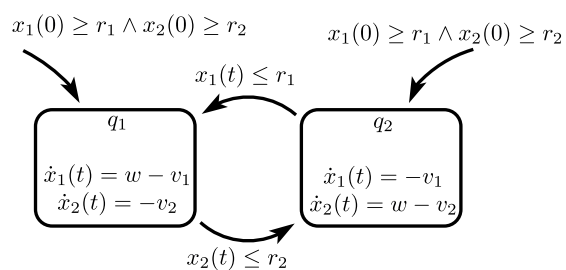


Figure 3.17: Hybrid automaton representing water tank system.

EXERCISES

- (b) A Zeno system may be **regularized** by ensuring that the time time between transitions is never less than some positive number ϵ . This can be emulated by inserting extra modes that in which the hybrid automaton dwells for time ϵ . Use regularization to make your model from part (a) non-Zeno. Again, plot x_1 and x_2 for the same length of time as in the first part. State the value of ϵ that you used.

Include printouts of your plots with your answer.

Chapter 4

Modeling Concurrent Systems

Contents

4.1 Synchronous Models	67
4.2 Time-Triggered Models	67
4.3 Event-Driven Models	67
4.4 Dataflow	67
4.5 Process Networks	68

This chapter, when written, will give an overview of models of computation that specifically model concurrent and timed models.

4.1 Synchronous Models

This section, when written, will give an overview of the **synchronous-reactive (SR)** model of computation. It will defined the **pre** operator, among other things.

4.2 Time-Triggered Models

4.3 Event-Driven Models

4.4 Dataflow

dataflow

4.5 Process Networks

Part II

Design of Embedded Systems

Chapter 5

Sensors and Actuators

Contents

5.1 Measuring Temperature	71
5.2 Measuring Motion	71
5.3 Measuring Time	72
5.4 Motion Control	72
5.4.1 Pulse Width Modulators	72

This chapter, when written, gives an overview of sensor technology with emphasis on how to model sensors.

5.1 Measuring Temperature

Measurement of temperature is central to HVAC systems, automotive engine controllers, overcurrent protection, and many industrial chemical processes. This section, when written, will give an overview of such technologies. (p. 28)

5.2 Measuring Motion

This section, when written, will give an overview of accelerometers, gyroscopes, and encoders.

5.3 Measuring Time

This section, when written, will give an overview of technologies for measuring the passage of time, including **crystal oscillators**.

5.4 Motion Control

5.4.1 Pulse Width Modulators

Chapter 6

Embedded Processors

Contents

6.1	Types of Processors	74
6.1.1	Microcontrollers	74
6.1.2	Programmable DSPs	75
6.1.3	Graphics Processors	81
6.2	Parallelism	81
6.2.1	Parallelism vs. Concurrency	81
6.2.2	Instruction-Level Parallelism	83
6.2.3	Pipelining	88
6.2.4	Multicore Architectures	92
	Exercises	94

In [general-purpose computing](#), the variety of processor architectures today is limited, with the Intel x86 architecture overwhelmingly dominating all. There is no such dominance in embedded computing. On the contrary, the variety of processors can be daunting to a system designer. Our goal in this chapter is to give the reader the tools and vocabulary to understand the options and to critically evaluate the properties of processors. We particularly focus on the mechanisms that provide concurrency and control over timing, because these issues loom large in the design of cyber-physical systems. (p. x)

When deployed in a product, embedded processors typically have a dedicated function. They control an automotive engine or measure ice thickness in the arctic. They are not asked to perform arbitrary functions with user-defined software. Consequently, the processors can be more specialized. Making them more specialized can bring enormous benefits. For example, they may consume far less energy, and consequently be usable with small batteries for long periods of time. Or they may include specialized hardware to perform operations that would be costly to perform on general-purpose hardware, such as image analysis.

When evaluating processors, it is important to understand the difference between an **instruction set architecture (ISA)** and a **processor realization** or a **chip**. The latter is a piece of silicon sold by a semiconductor vendor. The former is a definition of the instructions that the processor can execute and certain structural constraints (such as word size) that realizations must share. An ISA is an abstraction shared by many realizations. A single ISA may appear in many different chips, often made by different manufacturers, and often having widely varying performance profiles.

The advantage of sharing an ISA in a family of processors is that software tools, which are costly to develop, may be shared, and (sometimes) the same programs may run correctly on multiple chips. This latter property, however, is rather treacherous, since an ISA does not normally include any constraints on timing. Hence, although a program may execute logically the same way on multiple chips, the system behavior may be radically different when the processor is embedded in a cyber-physical system.

6.1 Types of Processors

As a consequence of the huge variety of embedded applications, there is a huge variety of processors that are used. They range from very small, slow, inexpensive, low-power devices, to high-performance, special-purpose devices. This section gives an overview of some of the alternatives.

6.1.1 Microcontrollers

A **microcontroller** (μC) is a small computer on a single integrated circuit consisting of a relatively simple **central processing unit (CPU)** combined with peripheral devices such as memories, I/O devices, and timers. By some accounts, more than half of all CPUs sold worldwide are microcontrollers. The simplest microcontrollers operate on 8-bit words and are suitable for applications that require small amounts of memory and simple logical functions (vs. performance-intensive arithmetic functions). They may consume extremely small amounts of energy, and often include a **sleep mode** that reduces the power consumption to nanowatts. Embedded applications such as sensor network nodes and surveillance devices have been demonstrated that can operate on a small battery for several years.

Microcontrollers can get quite elaborate. Distinguishing them from general-purpose processors can get difficult. The Intel Atom, for example, is a family of x86 CPUs used mainly in netbooks and other small mobile computers. Because these processors are designed to use relatively little energy without losing too much performance relative to processors used in higher-end computers, they are suitable for some embedded applications. AMD's Geode is another example of a processor near the

blurry boundary between general-purpose processors and microcontrollers.

6.1.2 Programmable DSPs

Many embedded applications do quite a bit of signal processing. A **signal** is a collection of sampled measurements of the physical world, typically taken at a regular rate called the sample rate. A motion control application, for example, may read position or location information from sensors at sample rates ranging from a few Hertz (Hz, or samples per second) to a few hundred Hertz. Audio signals are sampled at rates ranging from 8,000 Hz (or 8 kHz, the sample rate used in telephony for voice signals) to 44.1 kHz (the sample rate of CDs. Ultrasonic applications (such as medical imaging) and high-performance music applications may sample sound signals at much higher rates.

Microcontrollers

Most semiconductor vendors include one or more families of microcontrollers in their product line. Some of the architectures are quite old. The **Motorola 6800** and **Intel 8080** are 8-bit microcontrollers that appeared on the market in 1974. Descendants of these architectures survive today, for example in the form of the **Freescall 6811**. The **Zilog Z80** is a fully-compatible descendant of the 8080 that became one of the most widely manufactured and widely used microcontrollers of all time. A derivative of the Z80 is the Rabbit 2000 designed by Rabbit Semiconductor.

Another very popular and durable architecture is the **Intel 8051**, an 8-bit microcontroller developed by Intel in 1980. The 8051 **ISA** is today supported by many vendors, including at least Atmel, Infineon Technologies, Dallas Semiconductor, NXP, ST Microelectronics, Texas Instruments, and Cypress Semiconductor. The **Atmel AVR** 8-bit microcontroller, developed by Atmel in 1996, was one of the first microcontrollers to use on-chip **flash memory** for program storage. Although Atmel says AVR is not an acronym, it is believed that the **RISC** architecture was conceived by two students at the Norwegian Institute of Technology, Alf-Egil Bogen and Vegard Wollan, so it may have originated as Alf and Vegard's RISC.

Many 32-bit microcontrollers implement some variant an **ARM** instruction set, developed by ARM Limited. ARM originally stood for Advanced RISC Machine, and before that Acorn RISC Machine, but today it is simply ARM. ARM processors are widely used in mobile phones to realize the user interface functions, as well as in many other embedded systems. Semiconductor vendors license the instruction set from ARM Limited and produce their own chips. ARM processors are currently made by at least Alcatel, Atmel, Broadcom, Cirrus Logic, Freescale, LG, Marvell Technology Group, NEC, NVIDIA, NXP, Samsung, Sharp, ST Microelectronics, Texas Instruments, VLSI Technology, and Yamaha.

Other notable embedded microcontroller architectures include the **Motorola ColdFire** (later the Freescale ColdFire), the **Hitachi H8** and SuperH, the **MIPS** (originally developed by a team led by John Hennessy at Stanford University), the **PIC** (originally Programmable Interface Controller, from Microchip Technology), and the **PowerPC** (created in 1991 by an alliance of Apple, IBM, and Motorola).

(p. 113)

(p. 97)

(p. 84)

Video typically uses sample rates of 25 or 30 Hz for consumer devices to much higher rates for specialty measurement applications. Each sample, of course, contains an entire image (called a frame), which itself has many samples (called pixels) distributed in space rather than time. Software-defined radio applications have sample rates that can range from hundreds of kHz (for baseband processing) to several GHz (billions of Hertz). Other embedded applications that make heavy use of

The x86 Architecture

(p. 113) The dominant **ISA** for desktop and portable computers is known as the **x86**. This term originates with the Intel 8086, a 16-bit microprocessor chip designed by Intel in 1978. The 8086 was used in the original IBM PC, and the processor family has dominated the PC market ever since. Subsequent processors in this family were given names ending in “86,” and generally maintained backward compatibility. The Intel 80386 was the first 32-bit version of this instruction set, introduced in 1985. Today, the term “x86” usually refers to the 32-bit version, with 64-bit versions designated “x86-64.” The **Intel Atom**, introduced in 2008, is an x86 processor with significantly reduced energy consumption. Although it is aimed primarily at netbooks and other small mobile computers, it is also an attractive option for some embedded applications. The x86 architecture has also been implemented in processors from AMD, Cyrix, and several other manufacturers.

Programmable DSP Processors

Specialized computer architectures for signal processing have been around for quite some time (Allen, 1975). Single-chip DSP microprocessors first appeared in the early 1980s, beginning with the Western Electric DSP1 from Bell Labs, the S28211 from AMI, the TMS32010 from Texas Instruments, the uPD7720 from NEC, and a few others. Early applications of these devices included voiceband data modems, speech synthesis, consumer audio, graphics, and disk drive controllers. A comprehensive overview of DSP processor generations through the mid-1990s can be found in Lapsley et al. (1997).

(p. 98) A central characteristic of DSPs is a hardware multiply-accumulate unit, several variants of the **Harvard architecture** (to support multiple simultaneous operand and program fetches), and addressing modes supporting auto increment, circular buffers, and bit reversed addressing (the latter to support FFT calculation). Most support fixed-point data precisions of 16-24 bits, typically with much wider accumulators (40-56 bits) so that a large number of successive multiply-accumulate instructions can be executed without overflow. A few DSPs have appeared with floating point hardware, but these have not dominated the marketplace.

(p. 84) DSPs are all difficult to program compared to **RISC** architectures, primarily because of complex specialized instructions, a pipeline that is exposed to the programmer, and asymmetric memory architectures. Until the late 1990s, these devices were almost always programmed in assembly language. Even today, C programs make extensive use of libraries that are hand-coded in assembly language to take advantage of the most esoteric features of the architectures.

signal processing include interactive games, radar, sonar, and lidar imaging systems, video analytics (the extraction of information from video, for example for surveillance), driver assist systems for cars, medical electronics, and scientific instrumentation.

Signal processing applications all share certain characteristics. First, they deal with large amounts of data. The data may represent samples in time of a physical processor (such as samples of a wireless radio signal), samples in space (such as images), or both (such as video, radar, etc.). Second, they typically perform sophisticated mathematical operations on the data, including filtering, system identification, machine learning, and feature extraction. These operations are mathematically intensive.

Processors designed specifically to support numerically intensive signal processing applications are called **programmable DSP processors**, or **DSPs (digital signal processors)**, for short. To get some insight into the structure of such processors and the implications for the embedded software designer, it is worth understanding the structure of typical signal processing algorithms.

A canonical signal processing algorithm, used in some form in all of the above applications, is **finite impulse response (FIR)** filtering. The simplest form of this algorithm is straightforward, but has profound implications for hardware. In this simplest form, an input signal x consists of a very long sequence of numerical values, so long that for design purposes it should be considered infinite. Such an input can be modeled as a function $x: \mathbb{N} \rightarrow D$, where D is a set of values in some data type.¹ For example, D could be the set of all 16-bit integers, in which case, $x(0)$ is the first input value (a 16-bit integer), $x(1)$ is the second input value, etc. For mathematical convenience, we can augment this to $x: \mathbb{Z} \rightarrow D$ by defining $x(n) = 0$ for all $n < 0$. For each input value $x(n)$, an FIR filter must compute an output value $y(n)$ according to the formula,

$$y(n) = \sum_{i=0}^{N-1} a_i x(n-i) , \quad (6.1)$$

where N is the length of the FIR filter, and the coefficients a_i are called its **tap values**. You can see from this formula why it is useful to augment the domain of the function x , since the computation of $y(0)$, for example, involves values $x(-1)$, $x(-2)$, etc.

Example 6.1: Suppose $N = 4$ and $a_0 = a_1 = a_2 = a_3 = 1/4$. Then for all $n \in \mathbb{N}$,

$$y(n) = (x(n) + x(n-1) + x(n-2) + x(n-3))/4 .$$

Each output sample is the average the most recent four input samples. The structure of this computation is shown in Figure 6.1. In that figure, input values come in from the left and propagate down the **delay line**, which is tapped after each delay element. This structure is called a **tapped delay line**.

¹For a review of this notation, see Appendix A on page 153.

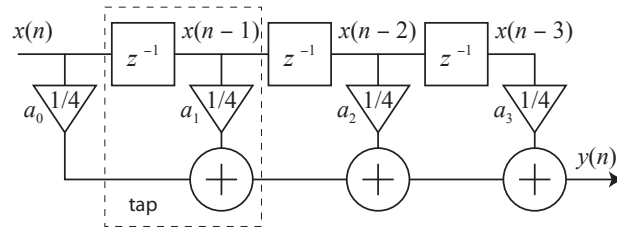


Figure 6.1: Structure of a tapped delay line implementation of the FIR filter of example 6.1. This diagram can be read as a dataflow diagram. For each $n \in \mathbb{N}$, each component in the diagram consumes one input value from each input path and produces one output value on each output path. The boxes labeled z^{-1} are **unit delays**. Their task is to produce on the output path the previous value of the input (or an initial value if there was no previous input). The triangles multiply their input by a constant, and the circles add their inputs.

The rate at which the input values $x(n)$ are provided and must be processed is called the **sample rate**. If you know the sample rate and N , you can determine the number of arithmetic operations that must be computed per second.

Example 6.2: Suppose that an FIR filter is provided with samples at a rate of 1 MHz (one million samples per second), and that $N = 32$. Then outputs must be computed at a rate of 1 MHz, and each output requires 32 multiplications and 31 additions. A processor must be capable of sustaining a computation rate of 63 million arithmetic operations per second to implement this application. Of course, to sustain the computation rate, it is necessary not only that the arithmetic hardware be fast enough, but also that the mechanisms for getting data in and out of memory and on and off chip be fast enough.

An image can be similarly modeled as a function $x: H \times V \rightarrow D$, where $H \subset \mathbb{N}$ represents the horizontal index, $V \subset \mathbb{N}$ represents the vertical index, and D is the set of all possible pixel values. A **pixel** (or picture element) is a sample representing the color and intensity of a point in an image. There are many ways to do this, but all use one or more numerical values for each pixel. The sets H and V depend on the **resolution** of the image.

Example 6.3: Analog television is steadily being replaced by digital formats such as ATSC, a set of standards developed by the Advanced Television Systems Committee. In the US, the vast majority of over-the-air NTSC transmissions were replaced with ATSC on June 12, 2009. ATSC supports a number of frame rates ranging from just below 24 Hz to 60 Hz and a number of resolutions. High-definition video under the ATSC standard

supports for example a resolution of 1080 by 1920 pixels at a frame rate of 30 Hz. Hence, $H = \{0, \dots, 1919\}$ and $V = \{0, \dots, 1079\}$. This resolution is called 1080p in the industry. Professional video equipment today goes up to four times this resolution (4320 by 7680). Frame rates can also be much higher than 30 Hz. Very high frame rates are useful for capturing extremely fast phenomena in slow motion.

For a grayscale image, a typical filtering operation will construct a new image y from an original image x according to the following formula,

$$\forall i \in H, j \in V, \quad y(i, j) = \sum_{n=i-N}^{i+N} \sum_{m=j-M}^{j+M} a_{n,m} x(i-n, j-m), \quad (6.2)$$

where $a_{n,m}$ are the filter coefficients. This is a two-dimensional FIR filter. Such a calculation requires defining x outside the region $H \times V$. There is quite an art to this (to avoid edge effects), but for our purposes here, it suffices to get a sense of the structure of the computation without being concerned for this detail.

A color image will have multiple **color channels**. These may represent luminance (how bright the pixel is) and chrominance (what the color of the pixel is), or it may represent colors that can be composed to get an arbitrary color. In the latter case, a common choice is an **RGBA** format, which has four channels representing red, green, blue, and the alpha channel, which represents transparency. For example, a value of zero for R, G, and B represents the color black. A value of zero for A represents fully transparent (invisible). Each channel also has a maximum value, say 1.0. If all four channels are at the maximum, the resulting color is a fully opaque white.

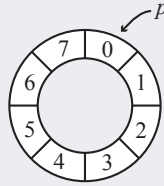
The computational load of the filtering operation in (6.2) depends on the number of channels, the number of filter coefficients (the values of N and M), the resolution (the sizes of the sets H and V), and the frame rate.

Example 6.4: Suppose that a filtering operation like (6.2) with $N = 1$ and $M = 1$ (minimal values for useful filters) is to be performed on high-definition video signal as in Example 6.3. Then each pixel of the output image y requires performing 9 multiplications and 8 additions. Suppose we have a color image with three channels (say, RGB, without transparency), then this will need to be performed 3 times for each pixel. Thus, each frame of the resulting image will require $1080 \times 1920 \times 3 \times 9 = 55,987,200$ multiplications, and a similar number of additions. At 30 frames per second, this translates into 1,679,616,000 multiplications per second, and a similar number of additions. Since this is about the simplest operation one may perform on a high-definition video signal, we can see that processor architectures handling such video signals must be quite fast indeed.

In addition to the large number of arithmetic operations, the processor has to handle the movement of data down the delay line, as shown later in Figure 6.1 (see box on page 80). By providing support for delay lines and multiply-accumulate instructions, as shown in Example 6.6, DSP processors can realize one tap of an FIR filter in one cycle. In that cycle, they multiply two numbers, add the result to an accumulator, and increment or decrement two pointers using modulo arithmetic.

Circular Buffers

(p. 77) An FIR filter requires a delay-line like that shown in Figure 6.1. A naive implementation would allocate an array in memory, and each time an input sample arrives, move each element in the array to the next higher location to make room for the new element in the first location. This would be enormously wasteful of memory bandwidth. A better approach is to use a **circular buffer**, where an array in memory is interpreted as having a ring-like structure, as shown below for a length-8 delay line:



Here, 8 successive memory locations, labeled 0 to 7, store the values in the delay line. A pointer p , initialized to location 0, provides access.

An FIR filter can use this circular buffer to implement the summation of (6.1). One implementation first accepts a new input value $x(n)$, and then calculates the summation backwards, beginning with the $i = N - 1$ term, where in our example, $N = 8$. Suppose that when the n^{th} input arrives, the value of p is some number $p_i \in \{0, \dots, 7\}$ (for the first input $x(0)$, $p_i = 0$). The program writes the new input $x(n)$ into the location given by p and then increments p , setting $p = p_i + 1$. All arithmetic on p is done modulo 8, so for example, if $p_i = 7$, then $p_i + 1 = 0$. The FIR filter calculation then reads $x(n - 7)$ from location $p = p_i + 1$ and multiplies it by a_7 . The result is stored in an **accumulator** register. It again increments p by one, setting it to $p = p_i + 2$. It next reads $x(n - 6)$ from location $p = p_i + 2$, multiplies it by a_6 , and adds the result to the accumulator (this explains the name “accumulator” for the register, since it accumulates the products in the tapped delay line). It continues until it reads $x(n)$ from location $p = p_i + 8$, which because of the modulo operation is the same location that the latest input $x(n)$ was written to, and multiplies that value by a_0 . It again increments p , getting $p = p_i + 9 = p_i + 1$. Hence, at the conclusion of this operation, the value of p is $p_i + 1$, which gives the location into which the next input $x(n + 1)$ should be written.

6.1.3 Graphics Processors

A **graphics processing unit (GPU)** is a specialized processor designed especially to perform the calculations required in graphics rendering. Such processors date back to the 1970s, when they were used to render text and graphics, to combine multiple graphic patterns, and to draw rectangles, triangles, circles, and arcs. Modern GPUs support 3D graphics, shading, and digital video. Dominant providers of GPUs today are Intel, NVIDIA and AMD.

Some embedded applications, particular games, are a good match for GPUs. Moreover, GPUs have evolved towards more general programming models, and hence have started to appear in other compute-intensive applications, such as instrumentation. GPUs are typically quite power hungry, and therefore today are not a good match for energy constrained embedded applications.

6.2 Parallelism

Most processors today provide various forms of parallelism. These mechanisms strongly affect the timing of the execution of a program, so embedded system designers have to understand them. This section provides an overview of the several forms and their consequences for system designers.

6.2.1 Parallelism vs. Concurrency

Concurrency is central to embedded systems. The word “concurrent” literally means “running together.” A computer program is said to be **concurrent** if different parts of the program *conceptually* execute simultaneously. A program is said to be **parallel** if different parts of the program *physically* execute simultaneously on distinct hardware (such as on multicore machines, servers in a server farm, or distinct microprocessors). (p. x)

Non-concurrent programs specify a *sequence* of instructions to execute. A programming language that expresses a computation as a sequence of operations is called an **imperative** language. C is an imperative language. When using C to write concurrent programs, we must step outside the language itself, typically using a **thread library**. A thread library uses facilities not provided by C, but rather provided by the operating system and/or the hardware. Java is a mostly imperative language extended with constructs that directly support threads. Thus, one can write concurrent programs in Java without stepping outside the language.

Every (correct) execution of a program in an imperative language must behave as if the instructions were executed exactly in the specified sequence. It is often possible, however, to execute instructions in parallel or in an order different from that specified by the program and still get behavior that matches what would have happened had they been executed in sequence.

Example 6.5: Consider the following C statements:

```
double pi, piSquared, piCubed;  
pi = 3.14159;  
piSquared = pi * pi ;  
piCubed = pi * pi * pi;
```

The last two assignment statements are independent, and hence can be executed in parallel or in reverse order without changing the behavior of the program. Had we written them as follows, however, they would no longer be independent:

```
double pi, piSquared, piCubed;  
pi = 3.14159;  
piSquared = pi * pi ;  
piCubed = piSquared * pi;
```

In this case, the last statement depends on the third statement in the sense that the third statement must complete execution before the last statement starts.

(p. 86)

A compiler may analyze the dependencies between operations in a program and produce parallel code, if the target machine supports it. This analysis is called **dataflow analysis**. Many micro-processors today support parallel execution, using multi-issue instruction streams or **VLIW** (very large instruction word) architectures. Processors with multi-issue instruction streams can execute independent instructions simultaneously. The hardware analyzes instructions on-the-fly for dependencies, and when there is no dependency, executes more than one instruction at a time. In the latter, VLIW machines have assembly-level instructions that specify multiple operations to be performed together. In this case, the compiler is usually required to produce the appropriate parallel instructions. In these cases, the dependency analysis is done at the level of assembly language or at the level of individual operations, not at the level of lines of C. A line of C may specify multiple operations, or even complex operations like procedure calls. In both cases (multi-issue and VLIW), an imperative program is analyzed for concurrency in order to enable parallel execution. The overall objective is to speed up execution of the program. The goal is improved **performance**, where the presumption is that finishing a task earlier is always better than finishing it later.

In the context of embedded systems, however, concurrency plays a part that is much more central than merely improving performance. Embedded programs interact with physical processes, and in the physical world, many activities progress at the same time. An embedded program often needs to monitor and react to multiple concurrent sources of stimulus, and simultaneously control multiple output devices that affect the physical world. Embedded programs are almost always concurrent programs, and concurrency is an intrinsic part of the logic of the programs. It is not just a way to

get improved performance. Indeed, finishing a task earlier is not necessarily better than finishing it later. *Timeliness* matters, of course; actions performed in the physical world often need to be done at the *right time* (neither early nor late). Picture for example an engine controller for a gasoline engine. Firing the spark plugs earlier is most certainly not better than firing them later. They must be fired at the *right time*.

Just as imperative programs can be executed sequentially or in parallel, concurrent programs can be executed sequentially or in parallel. Sequential execution of a concurrent program is done typically today by a **multitasking operating system**, which interleaves the execution of multiple tasks in a single sequential stream of instructions. Of course, the hardware may parallelize that execution if the processor has a multi-issue or VLIW architecture, getting an ironic path from concurrent program to sequential stream and back to concurrent program, where the latter translation is done to improve performance. These multiple translations greatly complicate the problem of ensuring that things occur at the *right time*. That problem is addressed in chapter 11.

This part studies techniques used to build concurrent programs. We proceed bottom up, discussing first in this chapter parallelism in the hardware and its implications for programmers. The next chapter covers the support for concurrent tasks at the level of the microprocessor hardware and instruction-set architecture (ISA), including interrupts, threads, and processes. The chapter after that discusses synchronization of concurrent tasks, including semaphores and mutual exclusion. We then conclude this part by discussing patterns of concurrency that provide more disciplined and understandable interactions between concurrent components by using message passing and more structured concurrent models of computation such as process networks and dataflow.

Parallelism in the hardware, the subject of this chapter, exists to improve performance for computation-intensive applications. From the programmer's perspective, concurrency arises as a consequence of the hardware designed to improve performance, not as a consequence of the application problem being solved. In other words, the application does not (necessarily) demand that multiple activities proceed simultaneously, it just demands that things be done very quickly. Of course, many interesting applications will combine both forms of concurrency, that of this chapter and that of the next.

We begin with a description of the sorts of algorithms found in compute-intensive embedded programs. The nature of these algorithms has a profound affect on the design of the hardware. We then continue with descriptions of distinct hardware approaches that deliver parallelism, namely instruction-level parallelism, pipelining, and multicore architectures. All have a strong influence on the programming models for embedded software. The reader may wish to review Appendix 7 for an overview of memory systems, which strongly influence how parallelism is handled.

6.2.2 Instruction-Level Parallelism

Achieving high performance demands parallelism in the hardware. Such parallelism can take two broad forms, multicore architectures, described later in Section 6.2.4, or **instruction-level paral-**

lelism (ILP), which is the subject of this section. A processor supporting ILP is able to perform multiple independent operations in each instruction cycle. We discuss four major forms of ILP, CISC instructions, subword parallelism, superscalar, and VLIW.

CISC Instructions

A processor with complex (and typically, rather specialized) instructions is called a **CISC** machine (**complex instruction set computer**). The philosophy behind such processors is distinctly different from that of **RISC** machines (**reduced instruction set computers**) (Patterson and Ditzel, 1980).

(p. 77) **DSPs** are typically CISC machines, and include instructions specifically supporting **FIR** filtering (and often other algorithms such as FFTs (fast Fourier transforms) and Viterbi decoding). In fact, to qualify as a DSP, a processor must be able to perform FIR filtering in one instruction cycle per tap.

Example 6.6: The Texas Instruments TMS320C54x family of DSP processors is intended to be used in power-constrained embedded applications that demand high signal processing performance, such as wireless communication systems and personal digital assistants (**PDA**s). The inner loop of the FIR computation of (6.1) is

```
1 RPT numberOfTaps - 1
2 MAC *AR2+, *AR3+, A
```

The first instruction illustrates the **zero-overhead loops** commonly found in DSPs. The instruction that comes after it will execute a number of times equal to one plus the argument of the **RPT** instruction. The **MAC** instruction is a **multiply-accumulate instruction**, also prevalent in DSP architectures. It has three arguments specifying the following calculation,

$$a := a + x * y,$$

(p. 80) where a is the contents of an **accumulator** register named **A**, and x and y are values found in memory. The addresses of these values are contained by auxiliary registers **AR2** and **AR3**. These registers are incremented automatically after the access. Moreover, these registers can be set up to implement **circular buffers**, as described in the box on page 80.

(p. 80) The C54x processor includes a section of on-chip memory that supports two accesses in a single cycle, and as long as the addresses refer to this section of the memory, the **MAC** instruction will execute in a single cycle. Thus, each cycle, the processor performs two memory fetches, one multiplication, one ordinary addition, and two (possibly modulo) address increments. All DSPs have similar capabilities.

CISC instructions can get quite esoteric.

Example 6.7: The coefficients of the FIR filter in (6.1) are often symmetric, meaning that N is even and

$$a_i = a_{N-i-1} .$$

The reason for this is that such filters have linear phase (intuitively, this means that symmetric input signals result in symmetric output signals, or that all frequency components are delayed by the same amount). In this case, we can reduce the number of multiplications by rewriting (6.1) as

$$y(n) = \sum_{i=0}^{(N/2)-1} a_i (x(n-i) + x(n-N+i+1)) .$$

The Texas Instruments TMS320C54x instruction set includes a `FIRS` instruction that functions similarly to the `MAC` in Example 6.6, but using this calculation rather than that of (6.1). This takes advantage of the fact that the C54x has two ALUs, and hence can do twice as many additions as multiplications. The time to execute an FIR filter now reduces to 1/2 cycle per tap.

CISC instruction sets have their disadvantages. For one, it is extremely challenging (perhaps impossible) for a compiler to make optimal use of such an instruction set. As a consequence, DSP processors are commonly used with code libraries written and optimized in assembly language.

In addition, CISC instruction sets can have subtle timing issues that can interfere with achieving hard-real-time deadlines. In the above examples, the layout of data in memory strongly affects execution times. Even more subtle, the use of zero-overhead loops (the `RPT` instruction above) can introduce some subtle problems. On the TI C54x, interrupts are disabled during repeated execution of the instruction following the `RPT`. This can result in unexpectedly long latencies in responding to interrupts.

Subword Parallelism

Many embedded applications operate on data types that are considerably smaller than the word size of the processor.

Example 6.8: In Examples 6.3 and 6.4, the data types are typically 8-bit integers, each representing a color intensity. The color of a pixel may be represented by three bytes in the RGB format. Each of the RGB bytes has a value ranging from 0 to 255 representing the intensity of the corresponding color. It would be wasteful of resources to use, say, a 64-bit ALU to process a single 8-bit number.

To support such data types, some processors support **subword parallelism**, where a wide ALU is divided into narrower slices enabling simultaneous arithmetic or logical operations on smaller words.

Example 6.9: Intel introduced subword parallelism into the widely used general purpose Pentium processor and called the technology MMX (Eden and Kagan, 1997). MMX instructions divide the 64-bit datapath into slices as small as 8 bits, supporting simultaneous identical operations on multiple bytes of image pixel data. The technology has been used to enhance the performance of image manipulation applications as well as applications supporting video streaming. Similar techniques were introduced by Sun Microsystems for SparcTM processors (Tremblay et al., 1996) and by Hewlett Packard for the PA RISC processor (Lee, 1996). Many processor architectures designed for embedded applications, including many DSP processors, also support subword parallelism.

(p. 77)

Superscalar

Superscalar processors use fairly conventional sequential instruction sets, but the hardware can simultaneously dispatch multiple instructions to distinct hardware units when it detects that such simultaneous dispatch will not change the behavior the program. That is, the execution of the program is identical to what it would have been if it had been executed in sequence. Such processors even support **out-of-order execution**, where instructions later in the stream are executed before earlier instructions. Superscalar processors have a significant disadvantage for embedded systems, which is that execution times may be extremely difficult to predict, and in the context of multitasking (interrupts and threads), may not even be repeatable. The execution times may be dependent on the exact timing of I/O operations, for example.

(p. 91)

(p. 114)

VLIW

Processors intended for embedded applications often use VLIW architectures instead of superscalar in order to get more repeatable and predictable timing. **VLIW (very large instruction word)** processors include multiple function units, like superscalar processors, but instead of dynamically determining which instructions can be executed simultaneously, each instruction specifies what each function unit should do in a particular cycle. That is, a VLIW instruction set combines multiple independent operations into a single instruction. Like superscalar architectures, these multiple operations are executed simultaneously on distinct hardware. Unlike superscalar, however, the order and simultaneity of the execution is fixed in the program rather than being decided on-the-fly. It is up to the programmer (working at assembly language level) or the compiler to ensure that the

simultaneous operations are indeed independent. In exchange for this additional complexity in programming, execution times become repeatable and (often) predictable.

Example 6.10: In Example 6.7, we saw the specialized instruction `FIRS` of the C54x architecture that specifies operations for two ALUs and one multiplier. This can be thought of as a primitive form of VLIW, but subsequent generations of processors are much more explicit about their VLIW nature. The Texas Instruments TMS320C55x, the next generation beyond the C54x, includes two multiply-accumulate units, and can support instructions that look like this:

```
1 MAC      *AR2+, *CDP+, AC0
2 :: MAC   *AR3+, *CDP+, AC1
```

Here, `AC0` and `AC1` are two accumulator registers and `CDP` is a specialized register for pointing to filter coefficients. The notation `::` means that these two instructions should be issued and executed in the same cycle. It is up to the programmer or compiler to determine whether these instructions can in fact be executed simultaneously. Assuming the memory addresses are such that the fetches can occur simultaneously, these two `MAC` instructions execute in a single cycle, effectively dividing in half the time required to execute an FIR filter.

For applications demanding higher performance still, VLIW architectures can get quite elaborate.

Example 6.11: The Texas Instruments c6000 family of processors have a VLIW instruction set. Included in this family are three subfamilies of processors, the c62x and c64x fixed-point processors and the c67x floating-point processors. These processors are designed for use in wireless infrastructure (cellular base stations, adaptive antennas, etc.), telecommunications infrastructure (voice over IP, video conferencing, etc.), and imaging applications (medical imaging, surveillance, machine vision or inspection, radar, etc.).

Example 6.12: The **TriMedia** processor family from NXP, is aimed at digital television, and can perform operations like that in (6.2) very efficiently. NXP Semiconductors used to be part of Philips, a diversified consumer electronics company that, among many other products, makes flat-screen TVs. The strategy in the TriMedia architecture is to make it easier for a compiler to generate efficient code, reducing the need for assembly-level programming (though even it includes specialized **CISC** instructions that are difficult for a compiler to exploit). It makes things easier for the compiler by having a larger register set

(p. 84)

(p. 84)
(p. 90)

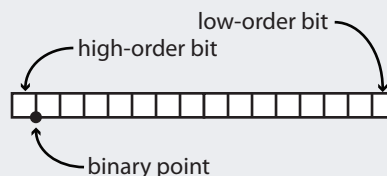
than is typical (128 registers) a RISC-like instruction set, where several instructions can be issued simultaneously, and hardware supporting IEEE 754 floating point operations.

6.2.3 Pipelining

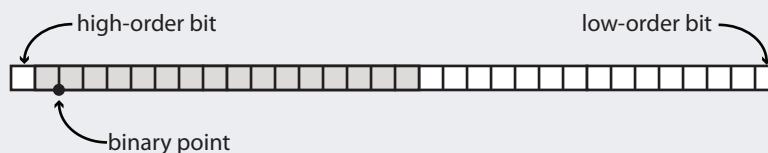
Most modern processors are pipelined. A simple five-stage pipeline for a 32-bit machine is shown in Figure 6.2. In the figure, the shaded rectangles are latches, which are clocked at processor clock rate. On each edge of the clock, the value at the input is stored in the latch register. The output is then held constant until the next edge of the clock, allowing the circuits between the latches to

Fixed-Point Numbers

Many embedded processors provide hardware only for integer arithmetic. Integer arithmetic, however, can be used for non-whole numbers, with some care. Given, say, a 16-bit integer, a programmer can *imagine* a **binary point**, which is like a decimal point, except that it separates bits rather than digits of the number. For example, a 16-bit integer can be used to represent numbers in the range -1.0 to 1.0 (roughly) by placing a (conceptual) binary point just below the high-order bit of the number, as shown below:



Without the binary point, a number represented by the 16 bits is a whole number $x \in \{-2^{15}, \dots, 2^{15} - 1\}$ (assuming the two's-complement binary representation, which has become nearly universal for signed integers). With the binary point, we *interpret* the 16 to represent a number $y = x/2^{15}$. Hence, y ranges from -1 to $1 - 2^{-15}$. This is known as a **fixed-point** number. The format of this fixed-point number can be written 1.15, indicating that there is one bit to the left of the binary point and 15 to the right. When two such numbers are multiplied at full precision, the result is a 32-bit number. The binary point is located as follows:



... Continued on page 89.

settle. This diagram can be viewed as [synchronous-reactive](#) model of the behavior of the processor, (p. 67)
and the latch can be understood as equivalent to the [pre](#) operator. (p. 67)

In the fetch (leftmost) stage of the pipeline, a **program counter (PC)** provides addresses to the instruction memory. The instruction memory provides encoded instructions, which in the figure are assumed to be 32 bits wide. In the fetch stage, the PC is incremented by 4 (bytes), to become the address of the next instruction, unless a conditional branch instruction is providing an entirely new address for the PC. The decode pipeline stage extracts register addresses from the 32-bit instruction and fetches the data in the specified registers from the register bank. The execute pipeline stage

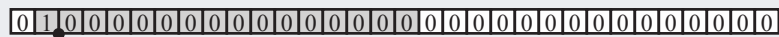
Fixed-Point Numbers (continued from page 88)

The location of the binary point follows from the **law of conservation of bits**. When multiplying two numbers with formats $n.m$ and $p.q$, the result has format $(n + p).(m + q)$. Processors often support such full-precision multiplications, where the result goes into an accumulator register that has at least twice as many bits as the ordinary data registers. To write the result back to a data register, however, we have to extract 16 bits from the 32 bit result. If we extract the shaded bits on page 89, then we preserve the position of the binary point, and the result still represents a number roughly in the range -1 to 1 .

There is a loss of information, however, when we extract 16 bits from a 32-bit result. First, there is a possibility of **overflow**, because we are discarding the high-order bit. Suppose the two numbers being multiplied are both -1 , which has binary representation in twos complement as follows:



When these two number are multiplied, the result has the following bit pattern:



which in twos complement, represents 1 , the correct result. However, when we extract the shaded 16 bits, the result is now -1 ! Indeed, 1 is not representable in the fixed-point format 1.15 , so overflow has occurred. Programmers must guard against this, for example by ensuring that all numbers are strictly less than 1 in magnitude, prohibiting -1 .

A second problem is that when we extract the shaded 16 bits from a 32-bit result, we discard 15 low-order bits. There is a loss of information here. If we simply discard the low-order 15 bits, the strategy is known as **truncation**. If instead we first add the following bit pattern the 32-bit result, then the result is known as **rounding**:



Rounding chooses the result that closest to the full-precision result, while truncation chooses the closest result that is smaller in magnitude.

DSP processors typically perform the above extraction with either rounding or truncation in hardware when data is moved from an accumulator to a general-purpose register or to memory.

operates on the data fetched from the registers or on the PC (for a computed branch) using an **arithmetic logic unit (ALU)**, which performs arithmetic and logical operations. The memory pipeline stage reads or writes to a memory location given by a register. The writeback pipeline stage stores results in the register file.

- (p. 77) DSP processors normally add an extra stage or two that performs a multiplication, provide separate ALUs for address calculation, and provide a dual data memory for simultaneous access to two operands (this latter design is known as a **Harvard architecture**). But the simple version in figure suffices to illustrate the issues that an embedded system designer faces.

Fixed-Point Arithmetic in C

Most C programmers will use `float` or `double` data types when performing arithmetic on non-whole numbers. However, many embedded processors lack hardware for floating-point arithmetic. Thus, C programs that use the `float` or `double` data types often result in unacceptably slow execution, since floating point must be emulated in software. Programmers are forced to use integer arithmetic to implement operations on numbers that are not whole numbers. How can they do that?

- (p. 88) First, a programmer can *interpret* a 32-bit `int` differently from the standard representation, using the notion of a **binary point**, explained in the boxes on pages 88 and 89. However, when a C program specifies that two `ints` be multiplied, the result is an `int`, not the full precision 64-bit result that we need. In fact, the strategy outlined on page 88, of putting one bit to the left of the binary point and extracting the shaded bits from the result, will not work, because most of the shaded bits will be missing from the result. For example, suppose we want to multiply 0.5 by 0.5. This number can be represented in 32-bit `ints` as follows:

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Without the binary point (which is invisible to C and to the hardware, residing only in the programmer's mind), this bit pattern represents the integer 2^{31} , a large number indeed. When multiplying these two numbers, the result is 2^{62} , which is not representable in an `int`. Typical processors will set an overflow bit in the processor status register (which the programmer must check) and deliver as a result the number 0, which is the low-order 32 bits of the product. To guard against this, a programmer can shift each 32 bit integer to the right by 16 bits before multiplying. In that case, the result of the multiply 0.5×0.5 is the following bit pattern:

0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- (p. 89) With the binary point as shown, this result is interpreted as 0.25, the correct answer. Of course, shifting data to the right by 16 bits discards the 16 low-order bits in the `int`. There is a loss of precision that amounts to **truncation**. The programmer may wish to round instead, adding the `int` 2^{15} to the numbers before shifting to the right 16 times. Floating-point data types make things easier. The hardware (or software) keeps track of the amount shifting required and preserves precision when possible. However, not all embedded processors with floating-point hardware conform with the **IEEE 754** standard. This can complicate the design process for the programmer, because numerical results will not match those produced by a desktop computer.

The portions of the pipeline between the latches operate in parallel. Hence, we can see immediately that there are simultaneously five instructions being executed, each at a different stage of execution. Given an instruction sequence A, B, C, D, E in a program, E is being fetched while D is being decoded, which occurs while C is being executed, B is reading or writing memory, and A is writing results to registers. Suppose that in such a sequence, instruction B reads from a register that instruction A writes to. Because of the pipeline, by the time A is writing to the register, B has long since past the stage where it reads from registers. The value that B reads will not be the value that A writes. This phenomenon is known as a **data hazard**, one form of **pipeline hazard**. Pipeline hazards are caused by the dashed lines in the figure.

Computer architects have tackled the problem of pipeline hazards in a variety of ways. The simplest technique is known as an **explicit pipeline**. In this technique, the pipeline hazard is simply documented, and the programmer (or compiler) must deal with it. For the example where B reads a register written by A , the compiler may insert three **no-op** instructions (which do nothing) between A and B to ensure that the write occurs before the read. These no-op instructions form a **pipeline bubble** that propagates down the pipeline.

A more elaborate technique is to provide **interlocks**. In this technique, the instruction decode hardware, upon encountering instruction B that reads a register written by A , will detect the hazard and delay the execution of B until A has completed the writeback stage. For this pipeline, B should be delayed by three clock cycles to permit A to complete (or by two cycles if slightly more complex **forwarding** logic is provided, which circumvents the read of the register bank). Interlocks therefore provide hardware that automatically inserts pipeline bubbles.

A still more elaborate technique is **out-of-order execution**, where hardware is provided that detects

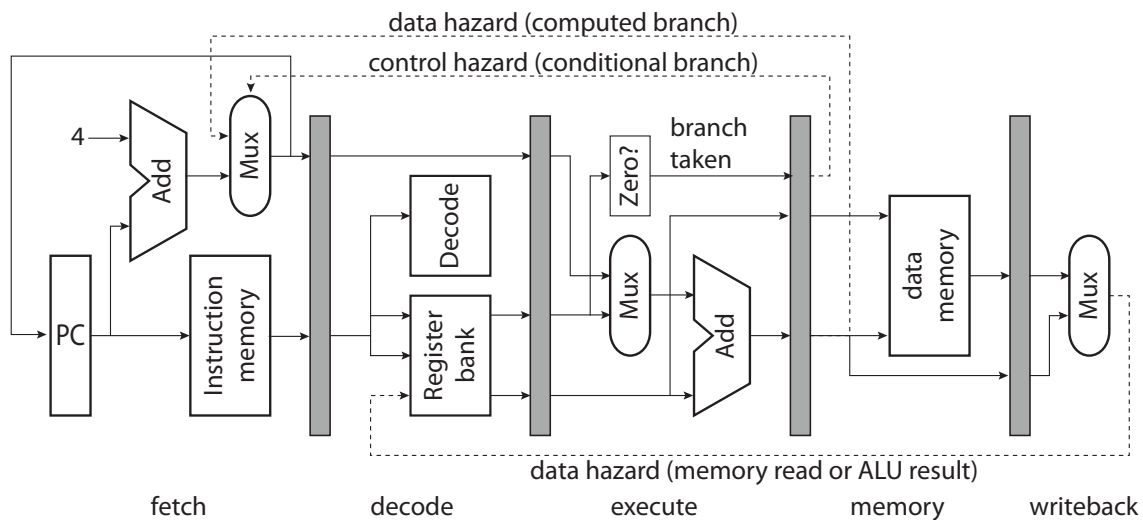


Figure 6.2: Simple pipeline (after [Patterson and Hennessey \(1996\)](#)).

a hazard, but instead of simply delaying execution of *B*, proceeds to fetch *C*, and if *C* does not read registers written by either *A* or *B*, and does not write registers read by *B*, then proceeds to execute *C* before *B*. This further reduces the number of pipeline bubbles.

Another form of pipeline hazard illustrated in Figure 6.2 is a **control hazard**. In the figure, a conditional branch instruction changes the value of the PC if a specified register has value zero. The new value of the PC is provided (optionally) by the result of an ALU operation. In this case, if *A* is a conditional branch instruction, then *A* has to have reached the memory stage before the PC can be updated. The instructions that follow *A* in memory will have been fetched and will be at the decode and execute stages already by the time it is determined that those instructions should not in fact be executed.

Like data hazards, there are multiple techniques for dealing with control hazards. A **delayed branch** simply documents the fact that the branch will be taken some number of cycles after it is encountered, and leaves it up to the programmer (or compiler) to ensure that the instructions that follow the conditional branch instruction are either harmless (like no-ops) or do useful work that does not depend on whether the branch is taken. An interlock provides hardware to insert bubbles as needed, just as with data hazards. In the most elaborate technique, **speculative execution**, hardware estimates whether the branch is likely to be taken, and begins executing the instructions it expects to execute. If its expectation is not met, then it undoes any side effects (such as register writes) that the speculatively executed instructions caused.

Except for explicit pipelines and delayed branches, all of these techniques introduce variability in the timing of execution of an instruction sequence. Analysis of the timing of a program can become extremely difficult when there is a deep pipeline with elaborate forwarding and speculation. Explicit pipelines are relatively common in DSP processors, which are often applied in contexts where precise timing is essential. Out-of-order and speculative execution are common in general-purpose processors, where timing matters only in an aggregate sense. An embedded system designer needs to understand the requirements of the application and avoid processors where the requisite level of timing precision is unachievable.

6.2.4 Multicore Architectures

A **multicore** machine is a combination of several processors on a single chip. Although multicore machines have existed since the early 1990s, they only recently have penetrated into general-purpose computing, accounting for much of the interest in them today. **Heterogeneous multicore** machines combine a variety of processor types on a single chip, vs. multiple instances of the same processor type.

Example 6.13: Texas Instruments **OMAP** (open multimedia application platform) architectures are widely used in cell phones, which normally combine one or more **DSP** processors with one or more processors that are closer in style to general-purpose processors.

(p. 77)

The DSP processors handle the radio, speech, and media processing (audio, images, and video). The other processors handle the user interface, database functions, networking, and downloadable applications. Specifically, the OMAP4440 includes a 1 GHz dual-core ARM Cortex processor, a C64x DSP, a GPU, and an image signal processor.

(p. 81)

For embedded applications, multicore architectures have a significant potential advantage over single-core architectures because real-time and safety-critical tasks can have a dedicated processor. This is the reason for the heterogeneous architectures used for cell phones, since the radio and speech processing functions are hard-real-time functions with considerable computational load. In such architectures, user applications cannot interfere with real-time functions.

This lack of interference is more problematic in general-purpose multicore architectures. It is common, for example, to use multi-level caches, where the second or higher level is shared among the cores. Unfortunately, such sharing makes it very difficult to isolate the real-time behavior of the programs on separate cores, since each program can trigger cache misses in another core. Such multi-level caches are not suitable for real-time applications.

(p. 99)

A very different type of multicore architecture that is sometimes used in embedded applications uses one or more **soft cores** together with custom hardware on a **field-programmable gate array (FPGA)**. FPGAs are chips whose hardware function is programmable using hardware design tools. Soft cores are processors implemented on FPGAs. The advantage of soft cores is that they can be tightly coupled to custom hardware more easily than off-the-shelf processors.

Exercises

1. Assuming fixed-point numbers with format 1.15 as described in the boxes on pages 88 and 89, show that the *only* two numbers that cause overflow when multiplied are -1 and -1 . That is, if either number is anything other than -1 in the 1.15 format, then extracting the 16 shaded bits in the boxes does not result in overflow.

Chapter 7

Memory Architectures

Contents

7.1	Volatility	96
7.1.1	Volatile Memory	96
7.1.2	Non-Volatile Memory	97
7.2	Memory Hierarchy	97
7.2.1	Memory Maps	98
7.2.2	Scratchpads and Caches	99
7.3	Memory Models	101
7.3.1	Memory Addresses	101
7.3.2	Stacks	102
7.3.3	Memory Protection Units	103
7.3.4	Dynamic Memory Allocation	103
7.3.5	Memory Model of C	104

Many processor architects argue that memory systems have more impact on overall system performance than data pipelines. This depends, of course, on the application, but for many applications it is true. The main sources of complexity in memory are three. First, many memory technologies are **volatile**, meaning that the contents of the memory is lost if power is lost. Most embedded systems need at least some non-volatile memory. Second, memory hierarchy is often needed because memories with larger capacity and/or lower power consumption are slower. To achieve reasonable performance at reasonable cost, faster memories must be mixed with slower memories. Third, the address space of a processor architecture is divided up to provide access to the various kinds of memory, to provide support for common programming models, and to designate addresses for interaction with devices other than memories, such as I/O devices. In this chapter, we discuss these three issues in order.

7.1 Volatility

In embedded systems, memory issues loom larger than in general purpose computing systems. A programmer may need to worry about whether data will persist when the power is turned off or a power-saving standby mode is entered. A memory whose contents are lost when the power is cut off is called a **volatile memory**. In this section, we discuss first volatile memory and then non-volatile memory.

7.1.1 Volatile Memory

The most tightly integrated volatile memory in a processor is the **register file**. Each register in the file stores a **word**, which is one byte on an 8-bit microcontroller, four bytes on a 32-bit microprocessor, and eight bytes on a 64-bit microprocessor. The cost and power consumption of the processor increases rapidly with word size. The register file is typically implemented as an **SRAM** (static RAM), often with multiple ports supporting simultaneous accesses. **RAM** (random access memory) is memory where any item (a byte, typically) can be accessed at any time by providing the address of that item, and the time of the access does not depend (much) on the address. For SRAM, reads and writes are both fast, usually matching the instruction cycle time of the processor.

(p. 113) In addition to the register file, a microcomputer typically includes an additional SRAM memory bank. The difference between this memory and the register file is that the addresses used to access the register file are directly encoded in the instructions of the instruction set architecture (**ISA**). The size of the register file, therefore, is limited more by the number of bits that can be devoted to such addresses than it is by the area occupied by the register file. The additional SRAM memory bank is accessed using the ISA's memory interface. The addresses are provided either in registers or as constants stored along with the program.

SRAM memory is relatively expensive in area, so the available SRAM is often not sufficient for useful applications. **DRAM** (dynamic random-access memory) is, like SRAM, a volatile memory, but it is both slower and much more compact than DRAM. As a consequence, much more memory can be included at modest cost in a design. The “dynamic” in DRAM refers to the fact that the memory is actually much more volatile than SRAM. SRAM will retain its values indefinitely as long as the power is on (hence the “static” in SRAM). DRAM, however, is forgetful. The data is stored in a capacitor charge, and the capacitor leaks. The charge must be refreshed. Typically, a manufacturer specifies that each memory location must be refreshed periodically, for example every 64 ms. Fortunately, a number of locations (a “row”) are refreshed together. The mere act of reading the memory will refresh the locations that are read (and locations on the same row). But since applications may not access all rows within the specified time interval, DRAM has to be used with a controller that ensures that all locations are refreshed sufficiently often to retain the data. This refresh introduces some timing variability in the memory accesses that may need to be taken into account in aggressive real-time designs.

7.1.2 Non-Volatile Memory

Embedded systems invariably need to store data even when the power is turned off. There are several options for this. One, of course, is to provide battery backup so that power is never lost. Batteries, however, wear out, and there are better options available, known collectively as **non-volatile memories**. An early form of non-volatile memory was **magnetic core memory** or just **core**, where an ferromagnetic ring was magnetized to store data. The term “core” persists in computing to refer to computer memories.

The most basic non-volatile memory today is **ROM** (read-only memory) or **mask ROM**, the contents of which is fixed at the factory. This can be useful for mass produced products that only need to have a program and constant data stored, and these data never change. Such programs are known as **firmware**, suggesting that they are not as “soft” as software. There are several variants of ROM that can be programmed in the field, and the technology has gotten good enough that these are almost always used today over mask ROM. **EEPROM**, electrically-erasable programmable ROM, comes in several forms, but all are capable of being written to. The write time is typically much longer than the read time, and the number of writes is limited during the lifetime of the device. A particularly useful form of EEPROM is flash memory. Flash is commonly used to store firmware and user data that needs to persist when the power is turned off.

Flash memories, invented by Dr. Fujio Masuoka at Toshiba around 1980, are a particularly convenient form of [non-volatile memory](#), but they present some interesting challenges for embedded systems designers. Typically, flash memories have reasonably fast read times, but not as fast as SRAM and DRAM, so frequently accessed data will typically have to be moved from the flash to RAM before being used by a program. The write times are much longer than the read times, and the total number of writes are limited, so these memories are not a substitute for working memory. (p. 97)

There are two types of flash memories, known as NOR and NAND flash. NOR flash has longer erase and write times, but it can be accessed like a RAM. NAND flash is less expensive and has faster erase and write times, but data must be read a block at a time, where a block is hundreds to thousands of bits. This means that from a system perspective it behaves more like a secondary storage device like a hard disk or optical media like CD or DVD. Both types of flash can only be erased and rewritten a bounded number of times, typically under 1,000,000 for NOR flash and under 10,000,000 for NAND flash, as of this writing.

The longer access times, limited number of writes, and block-wise accesses (for NAND flash), all complicate the software design problem for embedded system designers. These properties must be taken into account.

7.2 Memory Hierarchy

Many applications require substantial amounts of memory, more than what is available on-chip in a microcomputer. Many processors use a **memory hierarchy**, which combines different memory

(p. 96) technologies to increase the overall memory capacity at reasonable cost. Typically, a relatively small amount of on-chip **SRAM** will be used with a larger amount of off-chip **DRAM**. These can be further combined with a third level, such as disk drives, which have very large capacity, but lack random access and hence can be quite slow to read and write. When a non-RAM memory such as a disk is used to create the illusion of a larger memory, the scheme is called **virtual memory**. To understand how this works, it is useful to first understand how memory addresses are translated into physical locations in hardware.

7.2.1 Memory Maps

A **memory map** for a processor defines how addresses get mapped to hardware. The total size of the address space is constrained by the word width of the processor. A 32-bit processor, for example, can address 2^{32} locations, or 4 gigabytes (GB). An ARM Cortex™-M3 architecture, for example, has the memory map shown in figure 7.1. Other architectures will have other layouts, but the pattern is familiar.

Notice that this architecture separates addresses used for program memory (labeled A in the figure) from those used for data memory (B and D). This (rather typical) pattern allows these memories to be accessed via separate buses, permitting instructions and data to be fetched simultaneously. This effectively doubles the memory bandwidth. Such a separation of program memory from data memory is known as a **Harvard architecture**.

Any particular realization in silicon of this architecture is constrained by this memory map. For example, the Luminary Micro LM3S8962 controller, which includes an ARM Cortex™-M3 core, has 256 KB of on-chip flash memory, nowhere near the total of 0.5 GB that the architecture allows. This memory is mapped to addresses `0x00000000` through `0x0003FFFF`. The remaining addresses that the architecture allows for program memory, `0x00040000` through `0x1FFFFFFF` are “reserved,” meaning that they should not be used by a compiler targeting this particular device.

The LM3S8962 also has 64 KB of SRAM, mapped to addresses `0x20000000` through `0x2000FFFF`, a small portion of area D in the figure. It also includes a number of on-chip **peripherals**, which are devices that are accessed by the processor using some of the memory addresses in the range from

Harvard Architecture

(p. 4) The term “Harvard architecture” comes from the Mark I computer, which used distinct memories for program and data. The Mark I was made with electro-mechanical relays by IBM and shipped to Harvard in 1944. The machine stored instructions on punched tape and data in electro-mechanical counters. It was called the Automatic Sequence Controlled Calculator (ASCC) by IBM, and was devised by Howard H. Aiken to numerically solve **differential equations**. Rear Admiral Grace Murray Hopper of the United States Navy and funding from IBM were instrumental in making the machine a reality.

0x40000000 to 0x5FFFFFFF (area C). These include [timers](#), [GPIO](#), [UARTs](#), [ADCs](#), and other I/O devices. Each of these devices occupies a few of the memory addresses by providing **memory-mapped registers**. Some of these registers may be written to by the processor to configure and/or control the peripheral, or to provide data to be produced on an output. Some of the registers may be read to retrieve input data obtained by the peripheral. A few of the addresses in the private peripheral bus region are used to access the [interrupt controller](#). (p. 116)

The LM3S8962 is mounted on a printed circuit board that will provide additional devices such as [DRAM](#) data memory and additional external devices. As shown in figure 7.1, these will be mapped to memory addresses in the range from 0x60000000 to 0xDFFFFFFF (area E). For example, Luminary Micro sells a Stellaris® LM3S8962 Evaluation Board that includes no additional external memory, but does add a few external devices such as an LCD display, a MicroSD slot for additional flash memory, and a USB interface. (p. 96)

This leaves many memory addresses unused. ARM has introduced a clever way to take advantage of these unused addresses called **bit banding**, where some of the unused addresses can be used to access individual bits rather than entire bytes or words in the memory and peripherals. This makes certain operations more efficient, since extra instructions to mask the desired bits become unnecessary.

7.2.2 Scratchpads and Caches

Many embedded applications mix memory technologies. For example, a close memory (SRAM) is typically used to store working data temporarily while the program operates on it. If the close memory has a distinct set of addresses and the program is responsible for moving data into it or out of it to the distant memory, then it is called a **scratchpad** memory. If the close memory duplicates data in the distant memory with the hardware automatically handling the copying to and from, then it is called a **cache** memory. For embedded applications with tight real time constraints, cache memories present some formidable obstacles because their timing behavior can vary substantially in ways that are difficult to predict. On the other hand, manually managing the data in a scratchpad memory can be quite tedious for a programmer, and automatic compiler-driven methods for doing so are in their infancy.

As explained in Section 7.2.1, an architecture will typically support a much larger address space than what is needed to address the physical memory in an implementation of the processor. If the processor is equipped with a **memory management unit (MMU)**, then programs reference **logical addresses** and the MMU translates these to **physical addresses**. For example, using the memory map in figure 7.1, a [process](#) might be allowed to use logical addresses 0x60000000 to 0x9FFFFFFF (area D in the figure), for a total of 1 GB of addressable data memory. The MMU may implement a cache that uses however much physical memory is present in area B. When the program provides a memory address, the MMU determines whether that location is cached in area B, and if it is, translates the address and completes the fetch. If it is not, then we have a **cache miss**, and the MMU handles fetching data from the secondary memory (in area D) into the cache (area B). If the location is also not present in area D, then the MMU triggers a **page fault**, which can result (p. 133)

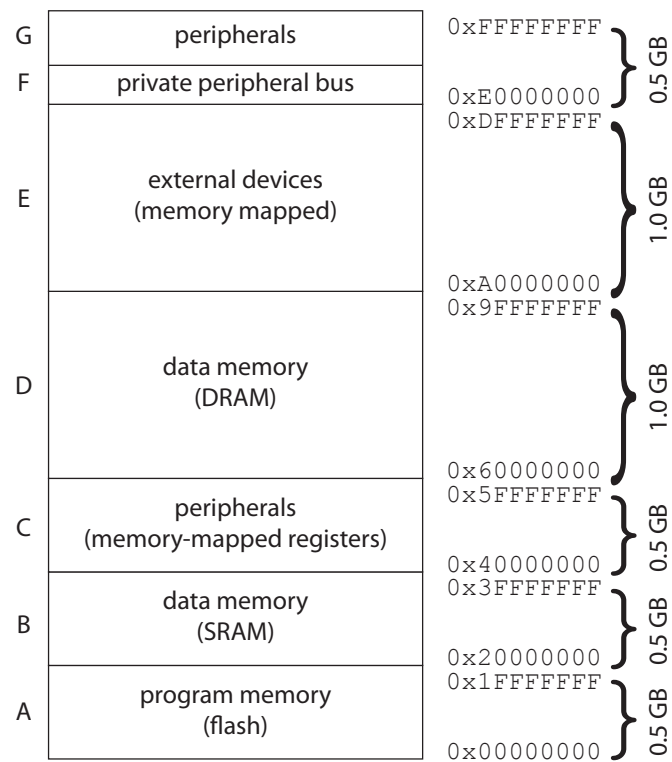


Figure 7.1: Memory map of an ARM Cortex™- M3 architecture.

in software handling movement of data from disk into the memory. Thus, the program is given the illusion of a vast amount of memory, with the cost that memory access times become quite difficult to predict.

Virtual memory and caches both enormously complicate the task of designing real-time software, since they can make memory access times highly variable. It is not uncommon for a memory access times to vary by a factor of 1000 or more, depending on how the logical addresses happen to be disbursed across the physical memories.

7.3 Memory Models

A **memory model** defines how memory is used by programs. The hardware, the operating system (if any), and the programming language and its compiler all contribute to the memory model. This section discusses a few of the common issues that arise with memory models.

7.3.1 Memory Addresses

At a minimum, a memory model defines a range of **memory addresses** accessible to the program. In C, these addresses are stored in **pointers**. In a **32-bit architecture**, memory addresses are 32-bit unsigned integers, capable of representing addresses 0 to $2^{32} - 1$, which is about four billion addresses. Each address refers to a byte (eight bits) in memory. The C **char** data type references a byte. The C **int** data type references a sequence of four bytes, able to represent integers from -2^{31} to $2^{31} - 1$. A **double** data type refers to a sequence of eight bytes encoded according to the IEEE floating point standard (IEEE 754).

Since a memory address refers to a byte, when writing a program that directly manipulates memory addresses, there are two critical compatibility concerns. The first is the **alignment** of the data. An int will typically occupy four consecutive bytes starting at an address that is a multiple of four. In hexadecimal notation these addresses always end in 0, 4, 8, or c.

The second concern is the byte order. The first byte (at an address ending in 0, 4, 8, or c), may represent the eight low order bits of the int (a representation called **little endian**), or it may represent the eight high order bits of the int (a representation called **big endian**). Unfortunately, although many data representation questions have become universal standards (such as the bit order in a byte), the byte order is not one those questions. Intel's x86 architectures and ARM processors, by default, use a little-endian representation, whereas IBM's PowerPC uses big endian. Some processors support both. Byte order also matters in network protocols, which generally use big endian.

The terminology comes from Gullivers Travels, by Jonathan Swift, where a royal edict in Lilliput requires cracking open one's soft-boiled egg at the small end, while in the rival kingdom of Blefuscu, inhabitants crack theirs at the big end.

7.3.2 Stacks

A **stack** is a region of memory that is dynamically allocated to the program in a last-in, first-out (**LIFO**) pattern. A **stack pointer** (typically a register) contains the memory address of the top of the stack. When an item is pushed onto the stack, the stack pointer is incremented and the item is stored at the new location referenced by the stack pointer. When an item is popped off the stack, the memory location referenced by the stack pointer is (typically) copied somewhere else (e.g. into a register) and the stack pointer is decremented.

Stacks are typically used to implement procedure calls. Given a procedure call in C, for example, the compiler produces code that pushes onto the stack the location of the instruction to execute upon returning from the procedure and the arguments to the procedure, and then sets the program counter equal to the location of the procedure code. When a procedure returns, the compiler pops all these things off the stack, retrieving finally the program location at which to resume execution.

For embedded software, it can be disastrous if the stack pointer is incremented beyond the memory allocated for the stack. Such a **stack overflow** can result in overwriting memory that is being used for other purposes, leading to unpredictable results. Bounding the stack usage, therefore, is an important part of the design problem. Note that this become particularly difficult with **recursive programs**, where a procedure calls itself. Embedded software designers often avoid using recursion to circumvent this difficulty.

More subtle errors can arise as a result of misuse or misunderstanding of the stack. Consider the following C program:

```
1  int* foo(int a) {
2      int b;
3      b = a * 10;
4      return &b;
5  }
6  int main(void) {
7      int* c;
8      c = foo(10);
9      ...
10 }
```

- (p. 104) The variable `b` is a **local variable**, with its memory on the stack. When the procedure returns, the variable `c` will contain a pointer to memory location *above the stack pointer*. The contents of that memory location will be overwritten when items are next pushed onto the stack. It is therefore incorrect for the procedure `foo` to return a pointer to `b`. By the time that pointer is de-referenced (i.e. if a line in `main` refers to `*c` after line 8), the memory location may contain something entirely different from what was assigned in `foo`. Unfortunately, C provides no protection against such errors.

7.3.3 Memory Protection Units

A key issue in systems that support multiple simultaneous tasks is preventing one task from disrupting the execution of another. This is particularly important in embedded applications that permit downloads of third party software, but it can also provide an important defense against software bugs in safety-critical applications.

Many processors provide **memory protection** in hardware. Tasks are assigned their own address space, and if a task attempts to access memory outside its own address space, a **segmentation fault** exception results. This will typically result in termination of the offending application.

7.3.4 Dynamic Memory Allocation

General-purpose software applications often have indeterminate requirements for memory, depending on parameters and/or user input. To support such applications, computer scientists have developed dynamic memory allocation schemes, where a program can at any time request of the operating system additional memory. The memory is allocated from a data structure known as a **heap**, which facilitates keeping track of which portions memory are in use by which application. Memory allocation occurs via an operating system call (such `malloc()` in C). When the program no longer needs access to memory that has been so allocated, it deallocates the memory (by calling `free()` in C).

Support for memory allocation often (but not always) includes garbage collection. For example, garbage collection is intrinsic in the Java programming language. A garbage collector is a task that runs either periodically or when memory gets tight that analyzes the data structures that a program has allocated and automatically frees any portions of memory that are no longer referenced within the program. When using a garbage collector, in principle, a programmer does not need to worry about explicitly freeing memory.

With or without garbage collection, it is possible for a program to inadvertently accumulate memory that is never freed. This is known as a memory leak, and for embedded applications, which typically must continue to execute for a long time, it can be disastrous. The program will eventually fail when physical memory is exhausted.

Another problem that arises with memory allocation schemes is memory fragmentation. This arises when a program chaotically allocates and deallocates memory in varying sizes. A fragmented memory has allocated and free memory chunks interspersed, and often the free memory chunks become too small to use. In this case, defragmentation is required.

Defragmentation and garbage collection are both very problematic for real-time systems. Straight-forward implementations of these tasks require all other executing tasks to be stopped while the defragmentation or garbage collection is performed. Implementations using such “stop the world” techniques can have substantial pause times, running sometimes for many milliseconds. Other tasks cannot execute during this time because references to data within data structures (pointers) are

inconsistent during the task. A technique that can reduce pause times is incremental garbage collection, which isolates sections of memory and garbage collects them separately. As of this writing, such techniques are experimental and not widely deployed.

7.3.5 Memory Model of C

C programs store data on the stack, on the heap, and in memory locations fixed by the compiler. Consider the following C program:

```
1  int a = 2;
2  void foo(int b, int* c) {
3      ...
4  }
5  int main(void) {
6      int d;
7      int* e;
8      d = ...;           // Assign some value to d.
9      e = malloc(sizeof(int)); // Allocate memory for e.
10     *e = ...;          // Assign some value to e.
11     foo(d, &e);
12     ...
13 }
```

(p. 102) In this program, the variable `a` is a **static variable** because it is declared outside any procedure definition. The compiler will assign it a fixed memory location. The variables `b` and `c` are **parameters**, which are allocated locations on the **stack** when the procedure `foo` is called. The variables `d` and `e` are **automatic variable** or **local variable**. They are declared within the body of a procedure (in this case, `main`). The compiler will allocate space on the stack for them.

(p. 103) When the procedure `foo` is called on line 11, the stack location for `b` will acquire a *copy* of the value of variable `d` assigned on line 8. This is an example of **pass by value**, where a parameter's value is copied onto the stack for use by the called procedure. The variable `e`, on the other hand, is allocated a memory location on the **heap**, and then it is **passed by reference**. Its *address* is stored in the stack location for `c`. If `foo` includes an assignment to `*c`, then after `foo` returns, that value can be read by referencing `*e`.

Chapter 8

Input and Output

Contents

8.1 I/O Hardware	106
8.1.1 General-Purpose Digital I/O	106
8.1.2 Serial Interfaces	110
8.1.3 Parallel Interfaces	113
8.2 Sequential Software in a Concurrent World	114
8.2.1 Interrupts and Exceptions	114
8.2.2 Atomicity	116
8.2.3 Interrupt Controllers	118
8.2.4 Modeling Interrupts	118
8.3 The Analog/Digital Interface	120
8.3.1 Digital to Analog and Analog to Digital Converters	121
8.3.2 Signal Conditioning	122
8.3.3 Sampling and Aliasing	125
Exercises	128

Because [cyber-physical systems](#) integrate computing and physical dynamics, the mechanisms in processors that support interaction with the outside world are central to any design. A system designer has to confront a number of issues. First, the mechanical and electrical properties of the interfaces are important. Incorrect use of parts may cause a system to malfunction or may reduce its useful lifetime. Second, in the physical world, many things happen at once. Software, by contrast, is mostly sequential. Reconciling these two disparate properties is a major challenge, and is often the biggest risk factor in the design of embedded systems. Incorrect interactions between sequential code and concurrent events in the physical world can cause dramatic system failures. Third, the physical world functions in a multidimensional continuum of time and space. It is an **analog** world. The world of software, however, is **digital**, and strictly quantized. Measurements of

(p. ix)

physical phenomena must be quantized in both magnitude and time before software can operate on them. And commands to the physical world that originate from software will also be intrinsically quantized. Understanding the effects of this quantization is essential. In this chapter, we deal with these three issues in order.

8.1 I/O Hardware

(p. 74) Embedded processors, be they [microcontrollers](#), [DSP](#) processors, or general-purpose processors, typically include a number of input and output (**I/O**) mechanisms on chip, exposed to designers as pins of the chip. In this section, we review some of the more common interfaces provided, illustrating their properties through the following running example.

(p. 113) **Example 8.1:** Figure 8.1 shows an evaluation board for the Luminary Micro Stellaris® microcontroller, which is an implementation of the ARM Cortex™- M3 32-bit [ISA](#). The microcontroller itself is in the center below the graphics display. Many of the pins of the microcontroller are available at the connectors shown on either side of the microcontroller and at the top and bottom of the board. Such a board would typically be used to prototype an embedded application, and in the final product it would be replaced with a custom circuit board that includes only the hardware required by the application. An engineer will develop software for the board using an integrated development environment (IDE) provided by the vendor and load the software onto [flash memory](#) to be inserted into the slot at the bottom of the board. Alternatively, software might be loaded onto the board through the [USB](#) interface at the top from the development computer.

(p. 97)

(p. 112)

The evaluation board in the above example is more than a processor since it includes a display and various hardware interfaces (switches and a speaker, for example). Such a board is often called a **single-board computer** or a **microcomputer board**. We next discuss a few of the interfaces provided by a microcontroller or single-board computer. For a more comprehensive description of the many kinds of I/O interfaces in use, we recommend [Valvano \(2007\)](#) and [Derenzo \(2003\)](#).

8.1.1 General-Purpose Digital I/O

Embedded system designers frequently need to connect specialized or custom hardware to embedded processors. Many embedded processors have a number **general-purpose I/O pins (GPIO)**, which enable the software to either read or write voltage levels representing a logical zero or one. If the processor **supply voltage** is V_{DD} , in **active high logic** a voltage close to V_{DD} represents a logical one, and a voltage near zero represents a logical zero. In **active low logic**, these interpretations are reversed.

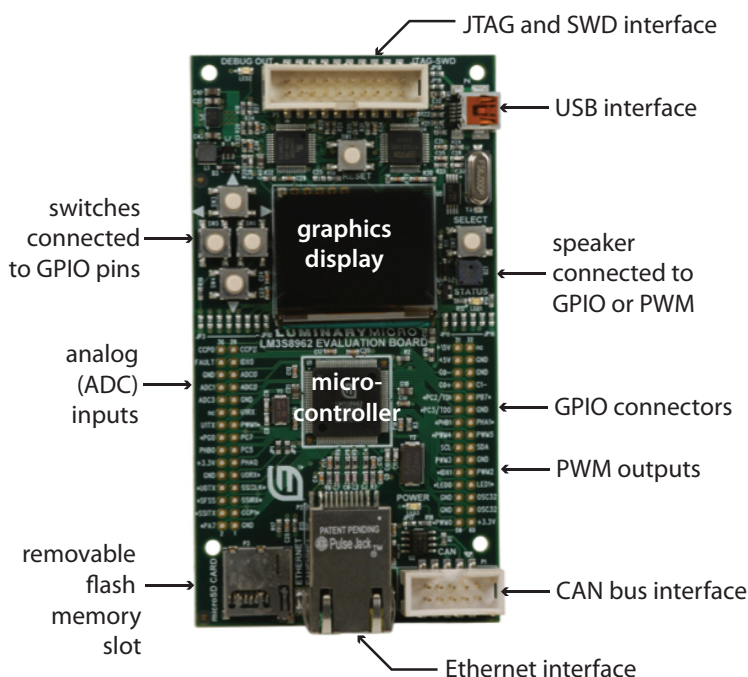


Figure 8.1: Stellaris®LM3S8962 evaluation board (Luminary Micro®, 2008a).

In many designs, GPIO pin may be configurable as an output by writing to a [memory-mapped register](#) (p. 99). This enables software to then write to another memory-mapped register to set the output voltage to be either high or low. By this mechanism, software can directly control external physical devices.

However, caution is in order. When interfacing hardware to GPIO pins, a designer needs to understand the specifications of the device. In particular, the voltage and current levels vary by device. If a GPIO pin produces an output voltage of V_{DD} when given a logical one, then the designer needs to know the current limitations before connecting a device to it. If a device with a resistance of R ohms is connected to it, for example, then **Ohm's law** tells us that the output current will be

$$I = V_{DD}/R .$$

It is essential to keep this current within specified tolerances. Going outside these tolerances could cause the device to overheat and fail. A **power amplifier** may be needed to deliver adequate current. An amplifier may also be needed to change voltage levels.

Example 8.2: The GPIO pins of the the Luminary Micro Stellaris®microcontroller shown in Figure 8.1 may be configured to source or sink varying amounts of current up

to 18 mA. There are restrictions on what combinations of pins can handle such relatively high currents. For example, [Luminary Micro® \(2008b\)](#) states “The high-current GPIO package pins must be selected such that there are only a maximum of two per side of the physical package ... with the total number of high-current GPIO outputs not exceeding four for the entire package.” Such constraints are designed to prevent overheating of the device.

In addition, it may be important to maintain **electrical isolation** between processor circuits and external devices. The external devices may have messy (noisy) electrical characteristics that will make the processor unreliable if the noise spills over into the power or ground lines of the processor. Or the external device may operate in a very different voltage or power regime compared to the processor. Isolation devices that may be used include opto isolators and transformers. The former convert an electrical signal in one electrical domain into light, and detect the light in the other electrical domain and convert it back to an electrical signal. The latter uses inductive coupling.

(p. 30) GPIO pins can also be configured as inputs, in which case software will be able to react to externally provided voltage levels. An input pin may be **Schmitt triggered**, for example, in which case they have [hysteresis](#), similar to the thermostat of Example 2.4. A Schmitt triggered input pin is less vulnerable to noise. It is named after Schmitt, Otto H. Otto H. Schmitt, who invented it in 1934 while he was a graduate student studying the neural impulse propagation in squid nerves.

Example 8.3: The GPIO pins of the microcontroller shown in Figure 8.1, when configured as inputs, are Schmitt triggered.

In many applications, several devices may share a single electrical connection. The designer must take care to ensure that these devices do not simultaneously drive the voltage of this single electrical connection to different values, resulting in a short circuit that can cause overheating and device failure.

Example 8.4: Consider a factory floor where several independent microcontrollers are all able to turn off a piece of machinery by asserting a logical zero on an output GPIO line. Such a design may provide additional safety because the microcontrollers may be redundant, so that failure of one does not prevent a safety-related shutdown from occurring. If all of these GPIO lines are wired together to a single control input of the piece of machinery, then we have to take precautions to ensure that the microcontrollers do not short each other out. This would occur if one microcontroller attempts to drive the shared line to a high voltage while another attempts to drive the same line to a low voltage.

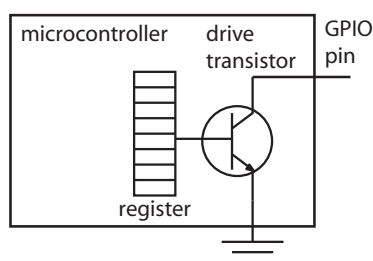


Figure 8.2: An open collector circuit for a GPIO pin.

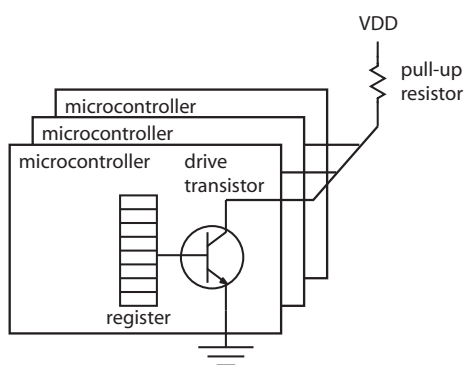


Figure 8.3: A number of open collector circuits wired together.

GPIO outputs may use **open collector** circuits, as shown in Figure 8.2. In such a circuit, writing a logical one into the (memory mapped) register turns on the transistor, which pulls the voltage on the output pin down to (near) zero. Writing a logical zero into the register turns off the transistor, which leaves the output pin unconnected, or “open.”

A number of open collector interfaces may be connected as shown in Figure 8.3. The shared line is connected to a **pull-up resistor**, which brings the voltage of the line up to V_{DD} when all the transistors are turned off. If any one transistor is turned on, then it will bring the voltage of the entire line down to (near) zero without creating a short circuit with the other GPIO pins. Logically, all registers must have zeros in them for the output to be high. If any one of the registers has a one in it, then the output will be low. Assuming **active high logic**, the logical function being performed is **NAND**, so such a circuit is called a **wired NAND**. By varying the configuration, one can similarly create wired OR or wired AND.

(p. 106)

The term “open collector” comes from the name for the terminal of a bipolar transistor. In CMOS technologies, this type of interface will typically be called an **open drain** interface. It functions essentially in the same way.

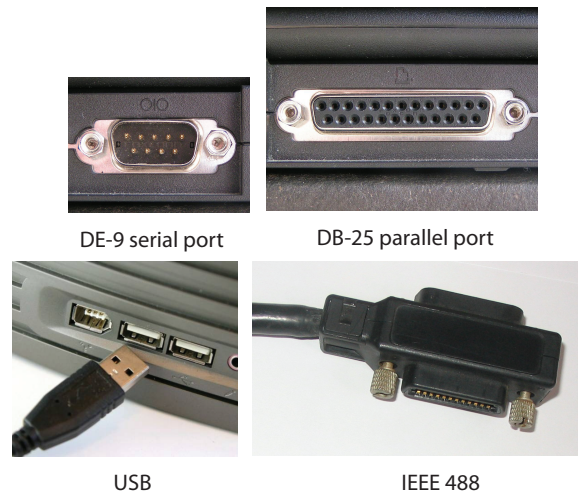


Figure 8.4: Connectors for serial and parallel interfaces.

Example 8.5: The GPIO pins of the microcontroller shown in Figure 8.1, when configured as outputs, may be specified to be open drain circuits. They may also optionally provide the pull-up resistor, which conveniently reduces the number of external discrete components required on a printed circuit board.

GPIO outputs may also be realized with **tristate** logic, which means that in addition to producing an output high or low voltage, the pin may be simply turned off. Like an open-collector interface, this can facilitate sharing the same external circuits among multiple devices. Unlike an open-collector interface, a tristate design can assert both high and low voltages, rather than just one of the two.

8.1.2 Serial Interfaces

One of the key constraints faced by embedded processor designers is the need to have physically small packages and low power consumption. A consequence is that the number of pins on the processor integrated circuit is limited. Thus, each pin must be used efficiently. In addition, when wiring together subsystems, the number of wires needs to be limited to keep the overall bulk and cost of the product in check. Hence, wires must also be used efficiently. One way to use pins and wires efficiently is to send information over them serially as sequences of bits. Such an interface is called a **serial interface**. A number of standards have evolved for serial interfaces. Using a standard interface (usually) enables connecting devices from different manufacturers.

A rather old but persistent standard, **RS-232**, standardized by the Electronics Industries Association (EIA), was first introduced in 1962 to connect teletypes to modems. This standard defines electrical signals and connector types; it persists because of its simplicity and because of continued prevalence of aging industrial equipment that uses it. The standard defines how one device can transmit a byte to another device asynchronously (meaning that the devices do not share a clock signal). On older PCs, an RS-232 connection may be provided via a DE-9 connector, as shown in Figure 8.4. A microcontroller will typically use a **universal asynchronous receiver/transmitter (UART)** to convert the contents of an 8-bit register into a sequence of bits for transmission over an RS-232 serial link.

For an embedded system designer, a major issue to consider is that RS-232 interfaces can be quite slow and may slow down the application software, if the programmer is not very careful.

Example 8.6: All variants of the [Atmel AVR](#) microcontroller include a UART that can be used to provide an RS-232 serial interface. To send a byte over the serial port, an application program may include the lines

(p. 75)

```
while (!(UCSR0A & 0x20));
UDR0 = x;
```

where `x` is a variable of type `uint8_t` (a C data type specifying an 8-bit unsigned integer). The symbols `UCSR0A` and `UDR0` are defined in header files provided in the AVR IDE. They are defined to refer to memory locations corresponding to [memory-mapped registers](#) in the AVR architecture.

(p. 106)

(p. 99)

The first line above executes an empty `while` loop until the serial transmit buffer is empty. The AVR architecture indicates that the transmit buffer is empty by setting the 6-th bit of the memory mapped register `UCSR0A` to 1. When that bit becomes 1, the expression `!(UCSR0A & 0x20)` becomes 0 and the `while` loop stops looping. The second line loads the value to be sent, which is whatever the variable `x` contains, into the memory-mapped register `UDR0`.

Suppose you wish to send a sequence of 8 bytes stored in an array `y`. You could do this with the C code

```
for(i = 0; i < 8; i++) {
    while (!(UCSR0A & 0x20));
    UDR0 = x[i];
}
```

How long would it take to execute this code? Suppose that the serial port is set to operate at 57600 baud, or bits per second (this is quite fast for an RS-232 interface). Then after loading `UDR0` with an 8-bit value, it will take $8/57600$ seconds or about 139 microseconds for the 8-bit value to be sent. Suppose that the frequency of the processor is operating at

18 MHz (relatively slow for a microcontroller). Then except for the first time through the `for` loop, each `while` loop will need to consume approximately 2500 cycles, during which time the processor is doing no useful work.

To receive a byte over the serial port, a programmer may use the following C code:

```
while (!(UCSR0A & 0x80));  
return UDR0;
```

In this case, the `while` loop waits until the UART has received an incoming byte. The programmer must ensure that there will be an incoming byte, or this code will execute forever. If this code is again enclosed in a loop to receive a sequence of bytes, then the `while` loop will need to consume a considerable number of cycles each time it executes.

For both sending and receiving bytes over a serial port, a programmer may use an [interrupt](#) instead to avoid having an idle processor that is waiting for the serial communication to occur. Interrupts will be discussed below.

(p. 114)

The RS-232 mechanism is very simple. The sender and receiver first must agree on a transmission rate (which due to the age of the standard is rather slow by modern standards). The sender initiates transmission of a byte with a **start bit**, which alerts the receiver that a byte is coming. The sender then clocks out the sequence of bits at the agreed-upon rate, following them by one or two **stop bits**. The receiver's clock resets upon receiving the start bit and is expected to track the sender's clock closely enough to be able to sample the incoming signal sequentially and recover the sequence of bits. There are many descendants of the standard that support higher rate communication, such as **RS-422**, **RS-423**, etc.

Newer devices designed to connect to personal computers typically use **universal serial bus (USB)** interfaces, standardized by a consortium of vendors. USB 1.0 appeared in 1996 and supports a data rate of 12 Mbits/sec. USB 2.0 appeared in 2000 and supports data rates up to 480 Mbits/sec. USB 3.0 appeared in 2008 and supports data rates up to 4.8 Gbits/sec.

USB is electrically simpler than RS-232 and uses simpler, more robust connectors, as shown in Figure 8.4. But the USB standard defines much more than electrical transport of bytes, and more complicated control logic is required to support it. Since modern peripheral devices such as printers, disk drives, and audio and video devices all include microcontrollers, supporting the more complex USB protocol is reasonable for these devices.

Another serial interface that is widely implemented in embedded processors is known as **JTAG** (joint test action group), or more formally as the IEEE 1149.1 standard test access port and boundary-scan architecture. This interface appeared in the mid 1980s to solve the problem that integrated circuit packages and printed circuit board technology had evolved to the point that testing circuits using electrical probes had become difficult or impossible. Points in the circuit that needed to be accessed became inaccessible to probes. The notion of a **boundary scan** allows the state of a logi-

cal boundary of a circuit (what would traditionally have been pins accessible to probes) to be read or written serially through pins that are made accessible. Today, JTAG ports are widely used to provide a debug interface to embedded processors, enabling a PC-hosted debugging environment to examine and control the state of an embedded processor. The JTAG port is used, for example, to read out the state of processor registers, to set breakpoints in a program, and to single step through a program.

There are several other serial interfaces in use today, including for example **I²C** (inter-integrated circuit), **SPI** (serial peripheral interface bus), **PCI Express** (peripheral component interconnect express), **MIDI** (musical instrument digital interface), and serial versions of **SCSI**. Each of these has its use. Also, network interfaces are typically serial. These are described in Chapter 10. (p. 113)

8.1.3 Parallel Interfaces

A serial interface sends or receives a sequence of bits sequentially over a single line. A **parallel interface** uses multiple lines to simultaneously send bits. Of course, each line of a parallel interface is also a serial interface, but the logical grouping and coordinated action of these lines is what makes the interface a parallel interface.

Historically, one of the most widely used parallel interfaces is the IEEE-1284 printer port, which on the IBM PC used a DB-25 connector, as shown in Figure 8.4. This interface originated in 1970 with the Centronics model 101 printer, and hence is sometimes called a Centronics printer port. Today, printers are typically connected using **USB** or wireless networks. (p. 112)

Another widespread parallel interface is **SCSI** (pronounced scuzzy, for small computer system interface), commonly used to connect hard drives and tape drives to computers. Recent variants of SCSI interfaces, however, depart from the traditional parallel interface to become serial interfaces. SCSI is an example of a peripheral bus architecture, used to connect computers to peripherals such as sound cards and disk drives. Other widely used standards include **ISA** (industry standard architecture, used in the ubiquitous IBM PC architecture), **PCI** (peripheral component interface), and **Parallel ATA** (advanced technology attachment).

A somewhat different kind of peripheral bus standard is **IEEE-488**, originally developed more than 30 years ago to connect automated test equipment to controlling computers. This interface was designed at Hewlett-Packard and is also widely known as **HP-IB** (Hewlett-Packard interface bus) and **GPIB** (general purpose interface bus).

With careful programming, a group of **GPIO** pins can be used together to realize a parallel interface. In fact, embedded system designers sometimes find themselves using GPIO pins to emulate an interface not supported directly by their hardware. (p. 106)

8.2 Sequential Software in a Concurrent World

As we saw in Example 8.6, when software interacts with the external world, the timing of the execution of the software may be strongly affected. Software is intrinsically sequential, typically executing as fast as possible. The physical world, however, is concurrent, with many things happening at once, and with the pace at which they happened determined by their physical properties. Bridging this mismatch in semantics is one of the major challenges that an embedded system designer faces. In this section, we discuss some of the key mechanisms for accomplishing this.

8.2.1 Interrupts and Exceptions

(p. 99) An **interrupt** is a mechanism for pausing execution of whatever a processor is currently doing
(p. 103) and executing a pre-defined code sequence called an **interrupt service routine (ISR)** or **interrupt handler**. Three kinds of events may trigger an interrupt. In the case of a **hardware interrupt**, some external hardware changes the voltage level on an interrupt request line. In the case of a **software interrupt**, the program that is executing triggers the interrupt by executing a special instruction or by writing to a **memory-mapped register**. A third variant is called an **exception**, where the interrupt is triggered by internal hardware that detects a fault, such as a **segmentation fault**.

For the first two variants, once the ISR completes, the program that was interrupted resumes where it left off. In the case of an exception, once the ISR has completed, the program that triggered the exception is not normally resumed. Instead, the program counter is set to some fixed location where, for example, the operating system may terminate the offending program.

Upon occurrence of an interrupt trigger, the hardware must first decide whether to respond. First, to respond, interrupts must be enabled. The mechanism for enabling or disabling interrupts varies by processor. Moreover, it may be that some interrupts are enabled and others are not. Interrupts and exceptions generally have priorities, and an interrupt will be serviced only if the processor is not already in the middle of servicing an interrupt with a higher priority. Typically, exceptions have the highest priority and are always serviced.

(p. 102) When the hardware decides to service an interrupt, it will usually first disable interrupts, push the current program counter and processor status register(s) onto the **stack**, and begin executing the instruction at a designated address, which will normally contain a jump to an ISR. The ISR must store on the stack the values currently in any registers that it will use, and restore their values before returning from the interrupt, so that the interrupted program can resume where it left off as if nothing had happened. Either the interrupt service routine or the hardware must also re-enable interrupts before returning from the interrupt.

(p. 116) **Example 8.7:** The ARM Cortex™- M3 is a 32-bit microcontroller used for example in industrial automation applications (ARM Limited, 2006). It includes a system **timer**

called SysTick. This timer can be used to trigger an ISR to execute every 1ms. Suppose for example that every 1ms we would like to count down from some initial count until the count reaches zero, and then stop counting down. The following C code defines an ISR that does this:

```
volatile uint timerCount = 0;
void countDown(void) {
    if (timerCount != 0) {
        timer_count--;
    }
}
```

Here, the variable timerCount is a [static variable](#), and it is decremented each time countDown() is invoked, until it reaches zero. We will specify below that this is to occur once per millisecond by registering countDown() as an ISR. The variable timerCount is marked with the C **volatile keyword**, which tells the compiler that the value of the variable will change at unpredictable times during execution of the program. This prevents the compiler from performing certain optimizations, such as caching the value of the variable in a register and reading it repeatedly. Using a C [API](#) provided by [Luminary Micro® \(2008c\)](#), we can specify that countDown() should be invoked as an interrupt service routine once per millisecond as follows:

```
SysTickPeriodSet(SysCtlClockGet() / 1000);
SysTickIntRegister(&countDown);
SysTickEnable();
SysTickIntEnable();
```

The first line sets the number of clock cycles between “ticks” of the SysTick timer. The timer will request an interrupt on each tick. SysCtlClockGet() is a library procedure that returns the number of cycles per second of the target platform’s clock (e.g., 50,000,000 for a 50 MHz part). The second line registers the ISR by providing a **function pointer** for the ISR (the address of the countDown() procedure). The third line starts the clock, enabling ticks to occur. The fourth line enables interrupts.

The timer service we have set up can be used, for example, to perform some function for two seconds and then stop. A program to do that is:

```
int main(void) {
    timerCount = 2000;
    ... initialization code from above ...
    while(timerCount != 0) {
        ... code to run for 2 seconds ...
    }
}
```

(p. 104)

(p. 133)

Processor vendors provide many variants of the mechanisms used in the previous example, so you will need to consult the vendor's documentation for the particular processor you are using. Since the code is not **portable** (it will not run correctly on a different processor), it is wise to isolate such code from your application logic and document carefully what needs to be re-implemented to target a new processor.

8.2.2 Atomicity

An interrupt service routine can be invoked between any two instructions of the main program (or between any two instructions of a lower priority ISR). One of the major challenges for embedded software designers is that reasoning about the possible interleavings of instructions can become extremely difficult. In the previous example, the interrupt service routine and the main program are interacting through a **shared variable**, namely `timerCount`. The value of that variable can change between any two **atomic operations** of the main program. Unfortunately, it can be quite difficult to know what operations are atomic. The term “atomic” comes from the Greek work for “indivisible,” and it is far from obvious to a programmer what operations are indivisible. If the programmer is writing assembly code, then it may be safe to assume that each assembly language instruction is atomic, but many ISAs include assembly level instructions that are not atomic.

(p. 113)

Example 8.8: The ARM instruction set includes a LDM instruction, which loads multiple registers from consecutive memory locations. It can be interrupted part way through the loads (ARM Limited, 2006).

At the level of a C program, it can be even more difficult to know what operations are atomic. Consider a single, innocent looking statement

Basics: Timers

Microcontrollers almost always include some number of peripheral devices called **timers**. A **programmable interval timer (PIT)**, the most common type, simply counts down from some value to zero. The initial value is set by writing to a **memory-mapped register**, and when the value hits zero, the PIT raises an interrupt request. By writing to a memory-mapped control register, a timer might be set up to trigger repeatedly without having to be reset by the software. Such repeated triggers will be more precisely periodic than what you would get if the ISR restarts the timer each time it gets invoked. This is because the time between when the count reaches zero in the timer hardware and the time when the counter gets restarted by the ISR is difficult to control and variable. For example, if the timer reaches zero at a time when interrupts happen to be disabled, then there will be a delay before the ISR gets invoked. It cannot be invoked before interrupts are re-enabled.

(p. 99)

```
timerCount = 2000;
```

On an 8-bit microcontroller, this statement may take more than one instruction cycle to execute (an 8-bit word cannot store both the instruction and the constant 2000; in fact, the constant alone does not fit in an 8-bit word). An interrupt could occur part way through execution of those cycles. Suppose that the ISR also writes to the variable `timerCount`. In this case, the final value of the `timerCount` variable may be composite of 8 bits set in the ISR and the remaining bits set by the above line of C, for example. The final value could be very different from 2000, and also different from the value specified in the interrupt service routine. Will this bug occur on a 32-bit microcontroller? The only way to know for sure is to fully understand the ISA and the compiler. In such circumstances, there is no advantage to having written the code in C instead of assembly language.

Bugs like this in a program are extremely difficult to identify and correct. Worse, the problematic interleavings are quite unlikely to occur, and hence may not show up in testing. For safety-critical systems, programmers have to make every effort to avoid such bugs. One way to do this is to build programs using higher-level concurrent models of computation, as discussed in Chapter 4. Of course, the implementation of those models of computation needs to be correct, but presumably, that implementation is constructed by experts in concurrency, rather than by application engineers.

When working at the level of C and ISRs, a programmer must carefully reason about the *order* of operations. Although many interleavings are possible, operations given as a sequence of C statements must execute in order (or more precisely, must behave as if they had executed in order, if [out-of-order execution](#) is possible).

(p. 91)

Example 8.9: In example 8.7, the programmer can rely on the statements within `main()` executing in order. Notice that in that example, the statement

```
timerCount = 2000;
```

appears before

```
SysTickIntEnable();
```

The latter statement enables the SysTick interrupt. Hence, the former statement cannot be interrupted by the SysTick interrupt.

8.2.3 Interrupt Controllers

An **interrupt controller** is the logic in the processor that handles interrupts. It supports some number of interrupts and some number of priority levels. Each interrupt has an **interrupt vector**, which is the address of an ISR or an index into an array called the **interrupt vector table** that contains the addresses of all the ISRs.

Example 8.10: The Luminary Micro LM3S8962 controller, shown in Figure 8.1, includes an ARM Cortex™ - M3 core microcontroller that supports 36 interrupts with eight priority levels. If two interrupts are assigned the same priority number, then the one with the lower vector will have priority over the one with the higher vector.

An interrupt is **asserted** to request that the corresponding ISR be executed. Mechanisms for asserting an interrupt are diverse. An external device could pull up or down the voltage of a pin to assert an interrupt. Or the external device or internal software could write to a designated memory address. Or the software could execute an interrupt instruction in the instruction set. Or the software could trigger a fault, such as a [segmentation fault](#).

The first of these, changing a voltage, may be either **level triggered** or **edge triggered**. For level-triggered interrupts, the hardware asserting the interrupt will typically hold the voltage on the line until it gets an acknowledgement, which indicates that the interrupt is being handled. For edge-triggered interrupts, the hardware asserting the interrupt changes the voltage for only a short time.

In both cases, [open collector](#) lines can be used so that the same the physical line can be shared among several devices (of course, the ISR will require some mechanism to determine which device asserted the interrupt, for example by reading a [memory-mapped register](#) in each device that could have asserted the interrupt).

Sharing interrupts among devices can be tricky, and careful consideration must be given to prevent low priority interrupts from blocking high priority interrupts. Asserting interrupts by writing to designated address on a bus has the advantage that the same hardware can support many more distinct interrupts, but the disadvantage that peripheral devices get more complex. They have to implement an interface to the memory bus.

8.2.4 Modeling Interrupts

The behavior of interrupts can be quite difficult to fully understand, and many catastrophic system failures are caused by unexpected behaviors. Unfortunately, the logic of interrupt controllers is often described in processor documentation very imprecisely, leaving many possible behaviors unspecified. One way to make this logic more precise is to model it as an [FSM](#). A partial specification,

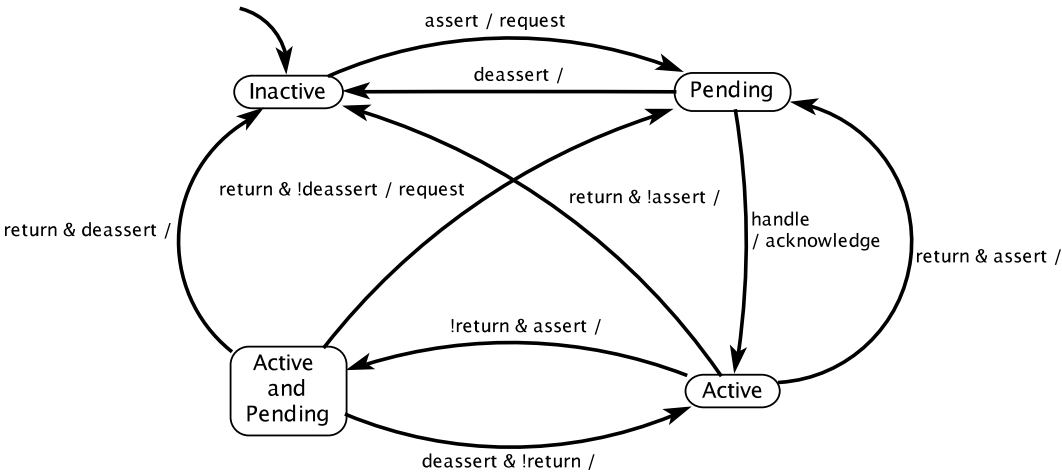


Figure 8.5: Protocol for an interrupt controller, showing the state of one interrupt.

for example, is given in Figure 8.5. This figure shows the state of a single interrupt. The four states have the following meaning

Inactive	no interrupt request pending or active
Pending	interrupt has been requested but not serviced
Active	interrupt service routine is executing
Active and Pending	interrupt has been requested while active

The input and output symbols of this FSM are:

Input Symbols	
assert	interrupt request
deassert	cancel a previous interrupt request
handle	message from an arbiter to handle this interrupt
return	return from interrupt
Output Symbols	
acknowledge	indicate that this interrupt is being handled
request	message to an arbiter requesting handling of interrupt

The initial state is *Inactive*. An *assert* input requests an interrupt, and the state changes to *Pending*. An arbiter (not shown, see Problem 4) reacts to this request by (possibly) sending back a *handle* signal. That signal indicates that the arbiter has chosen this interrupt to handle. Upon receiving the *handle* input, the FSM in Figure 8.5 sends *acknowledge* back to the originator of the *assert* and transitions to the *Active* state.

- (p. 27) As in Section 2.3, the **guards** on the transition are given as logical expressions, a shortcut for a subset of the input symbols. Specifically, the input symbols are the set

$$I = \{assert, deassert, handle, return\}$$

and the set of all possible inputs is the powerset 2^I . The guard “*assert*” is a shorthand for the set

$$\{i \in 2^I : assert \in i\}.$$

That is, the guard *assert* triggers the transition whenever the input includes the symbol *assert*. The guard “*return & !assert*” is shorthand for the set

$$\{i \in 2^I : return \in i \wedge assert \notin i\}.$$

Notice that in Figure 8.5, if the current state is *Inactive* and the input includes both *assert* and *deassert*, then the state changes to *Pending*. The *deassert* is ignored. Unfortunately, documentation provided by vendors is often ambiguous about what happens in such unexpected circumstances. The FSM representation enables us to be precise.

While in the *Pending* state, if the input includes *handle*, then the transition to *Active* is enabled. The FSM sends out an *acknowledge* signal (this signal may cause the hardware peripheral requesting the interrupt to let float the request line) and invokes the ISR. If *deassert* occurs before *handle*, then the request is canceled before being serviced. What happens if *deassert* and *handle* occur simultaneously? This FSM unambiguously states that the result is nondeterminate. The request may be canceled or serviced. Both reactions conform to the specification in Figure 8.5.

Each interrupt can be represented by a state machine like that in Figure 8.5. An arbiter will be responsible for deciding whether to handle a particular interrupt request. That decision will be based on what interrupts are enabled and what interrupts (if any) are currently pending or active. Processor architectures usually provide mechanisms to prioritize these requests and to temporarily mask (disable) only certain interrupts. These mechanisms can be quite complex. Software that uses these mechanisms to provide I/O to an external device is called a **device driver**. Writing device drivers that are correct and robust is a challenging engineering task requiring a deep understanding of the architecture and considerable skill reasoning about concurrency.

8.3 The Analog/Digital Interface

Cyber-physical systems typically require that measurements of physical properties be taken and processed by computers that then issue commands to actuators to have some effect on the physical world. At the boundary of the cyber and physical worlds, measurements must be converted to digital data, and digital data must be converted to analog effects on the physical world. Issues that arise in these conversions include distortion due to quantization and sampling and dealing with noise in the physical environment. We discuss those issues in this section.

8.3.1 Digital to Analog and Analog to Digital Converters

An analog signal varies continuously in both time and amplitude. Mathematically, such a signal might be represented as a function $x: \mathbb{R} \rightarrow \mathbb{R}$, where the domain represents time and the codomain represents amplitude. A simple conversion of such a signal to digital form is performed by an **analog comparator**, which compares the value against a threshold and produces a binary value, zero or one. For example, we could define a function $q: \mathbb{R} \rightarrow \{0, 1\}$ by

$$q(t) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

for all $t \in \mathbb{R}$. Such a signal is discrete in amplitude, but still has a continuous time base. This signal is **quantized**, in this case rather harshly so that the quantized signal can only take on one of two values. The signal q can be viewed as an approximation of the signal x , albeit not necessarily a very good approximation.

Suppose that we set up a software system to examine this signal at regularly spaced times called the **sample period**. For example, given an analog circuit that produces the signal q as an input to a **GPIO** pin, we could set up a timer **interrupt** to regularly examine the value at that pin and convert it to a boolean value in software. The resulting signal is a function $y: \mathbb{Z} \rightarrow \{0, 1\}$ given by

(p. 106)

$$y(n) = q(nT)$$

for all $n \in \mathbb{Z}$, where T is the sample period. This is a **digital signal** because it is discrete in both time and amplitude.

A better approximation of the original signal x might allow more than two possible values for each sample. The values could be, for example, those that can be represented by some fixed-point numbering scheme as explained in the box on page 88. An **analog to digital converter (ADC)** is a hardware device that performs just such a conversion. It has two key parameters, the sample period T and the number of bits b in the digital representation of the results. For the analog comparator discussed above, $b = 1$. The choice of b and T represents a tradeoff between cost and precision.

Example 8.11: For audio signals from a compact disc (CD), $T = 1/44,100$ and $b = 16$. This sample period is (barely) adequate to accurately represent frequency ranges audible to the human ear. And 16 bits is (barely) adequate to reduce **quantization noise** (the distortion resulting from quantization) to inaudible levels.

For a given value b , there are 2^b possible values, so having a larger value for b results in a closer approximation to the analog signal x . Moreover, the smaller T is, the more temporal detail of the signal that is preserved in its digital representation. In practice, the larger b is, the more difficult it is to make T small. Thus, high-precision ADCs (those with large b) tend to support slower sampling rates (larger T).

(p. 78)

Example 8.12: The ATSC digital video coding standard includes a format where the frame rate is 30 frames per second and each frame contains $1080 \times 1920 = 2,073,600$ pixels. An ADC that is converting one color channel to a digital representation must therefore perform $2,073,600 \times 30 = 62,208,000$ conversions per second, which yields a sample period T of approximately 16 nsec. With such a short sample period, increasing b becomes very expensive. For video, a choice of $b = 8$ is generally adequate to yield good visual fidelity and can be realized at reasonable cost.

A **digital to analog converter (DAC)** performs the converse conversion. Given a sampling period T and a sequence of digital values, each with b bits, it produces a continuous-time signal (a voltage vs. time) that, were it to be sampled by an ADC with parameters T and b would yield the same digital sequence (or, at least, a similar digital sequence).

The design of ADC and DAC hardware is itself quite an art. The effects of choices of T and b are also quite nuanced. Considerable expertise in signal processing is required to fully understand the implications of choices. In the remainder of this section, we give only a cursory view of this rather sophisticated topic. We begin with a discussion of how to mitigate the affect of noise in the environment, showing the rather intuitive result that it is beneficial to filter out frequency ranges that are not of interest. We then follow with a section on how to understand the effects of sampling, reviewing the Nyquist-Shannon sampling theorem, which gives us the guideline that we should sample continuous time signals at rates at least twice as high as the largest frequency of interest.

8.3.2 Signal Conditioning¹

Sensors convert physical measurements into data. Invariably, they are far from perfect, in that the data they yield gives information about the physical phenomenon that we wish to observe and other phenomena that we do not wish to observe. Removing or attenuating the effects of the phenomena we do not wish to observe is called **signal conditioning**.

Suppose that a sensor yields a continuous-time signal x . We model it as a sum of a **desired part** x_d and an **undesired part** x_n ,

$$x(t) = x_d(t) + x_n(t). \quad (8.1)$$

The undesired part is called **noise**. To condition this signal, we would like to remove or reduce x_n without affecting x_d . In order to do this, of course, there has to be some meaningful difference between x_n and x_d . Often, the two parts differ considerably in their frequency content.

¹This section requires a background in signals and systems at the level typically covered in a sophomore or junior engineering course.

Example 8.13: Consider using an accelerometer to measure the orientation of a slowly moving object. The accelerometer reacts to changes in orientation because they change the direction of the gravitational field with respect to its axis. But it will also report acceleration due to vibration. Let x_d be the signal due to orientation and x_n be the signal due to vibration. We will assume that x_n has higher frequency content than x_d . Thus, by frequency-selective filtering, we can reduce the effects of vibration.

Analysis of the degree to which frequency selective filtering helps requires having a model of both the desired signal x_d and the noise x_n . Reasonable models are usually statistical, and analysis of the signals requires using the techniques of random processes. Although such analysis is beyond the scope of this text, we can gain insight that is useful in many practical circumstances through a purely deterministic analysis.

Our approach will be to condition the signal $x = x_d + x_n$ by filtering it with an LTI system S called a **conditioning filter**. Let the output of the conditioning filter be given by (p. 12)

$$y = S(x) = S(x_d + x_n) = S(x_d) + S(x_n),$$

where we have used the linearity assumption on S . Let the error signal be defined to be

$$r = y - x_d.$$

This signal tells us how far off the filtered output is from the desired signal. The **energy** in the signal r is defined to be

$$\|r\|^2 = \int_{-\infty}^{\infty} r^2(t) dt.$$

We define the **signal to noise ratio (SNR)** to be

$$SNR = \frac{\|x_d\|^2}{\|r\|^2}.$$

Combining the above definitions, we can write this as

$$SNR = \frac{\|x_d\|^2}{\|S(x_d) - x_d + S(x_n)\|^2}. \quad (8.2)$$

It is customary to give SNR in **decibels**, written **dB**, defined as follows,

$$SNR_{dB} = 10 \log_{10}(SNR).$$

Note that for typical signals in the real world, the energy is effectively infinite if the signal goes on forever. A statistical model, therefore, would use the **power**, defined as the expected energy per

unit time. But since we are avoiding using statistical methods here, we will stick to energy as the criterion.

A reasonable design objective for a conditioning filter is to maximize the SNR. Of course, it will not be adequate to use a filter that maximizes the SNR only for particular signals x_d and x_n . We cannot know when we design the filter what these signals are, so the SNR needs to be maximized in expectation. That is, over the ensemble of signals we might see when operating the system, weighted by their likelihood, the expected SNR should be maximized.

Although determination of this filter requires statistical methods beyond the scope of this text, we can draw some intuitively appealing conclusions by examining (8.2). The numerator is not affected by S , so we can ignore it and minimize the denominator. It is easy to show that the denominator is bounded as follows,

$$\|r\|^2 \leq \|S(x_d) - x_d\|^2 + \|S(x_n)\|^2 \quad (8.3)$$

which suggests that we may be able to minimize the denominator by making $S(x_d)$ close to x_d (i.e. make $\|S(x_d) - x_d\|^2$ small) while making $\|S(x_n)\|^2$ small. That is, the filter S should do minimal damage to the desired signal x_d while filtering out as much as possible of the noise. This, of course, is obvious.

As illustrated in Example 8.13, x_d and x_n often differ in frequency content. We can get further insight using **Parseval's theorem**, which relates the energy to the Fourier transform,

$$\|r\|^2 = \int_{-\infty}^{\infty} (r(t))^2 dt = \frac{1}{2\pi} \int_{-\infty}^{\infty} |R(\omega)|^2 d\omega = \frac{1}{2\pi} \|R\|^2$$

where R is the Fourier transform of r .

The filter S is an LTI system. It is defined equally well by the function $S: (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$, by its impulse response $h: \mathbb{R} \rightarrow \mathbb{R}$, a continuous-time signal, or by its **transfer function** $H: \mathbb{R} \rightarrow \mathbb{C}$, the Fourier transform of the impulse response. Using the transfer function and Parseval's theorem, we can write

$$SNR = \frac{\|X_d\|^2}{\|HX_d - X_d + HX_n\|^2}, \quad (8.4)$$

where X_d is the Fourier transform of x_d and X_n is the Fourier transform of x_n . In Problem 6, we explore a very simple strategy that chooses the transfer function so that $H(\omega) = 1$ in the frequency range where x_d is present, and $H(\omega) = 0$ otherwise. This strategy is not exactly realizable in practice, but an approximation of it will work well for the problem described in Example 8.13.

Note that it is easy to adapt the above analysis to discrete-time signals. If $r: \mathbb{Z} \rightarrow \mathbb{R}$ is a discrete-time signal, its energy is

$$\|r\|^2 = \sum_{n=-\infty}^{\infty} (r(n))^2.$$

If its discrete-time Fourier transform (**DTFT**) is R , then Parseval's relation becomes

$$\|r\|^2 = \sum_{n=-\infty}^{\infty} (r(n))^2 = \frac{1}{2\pi} \int_{-\pi}^{\pi} |R(\omega)|^2 d\omega.$$

Note that the limits on the integral are different, covering one cycle of the periodic DTFT. All other observations above carry over unchanged.

8.3.3 Sampling and Aliasing²

Almost every embedded system will sample and digitize sensor data. In this section, we review the phenomenon of aliasing. We use a mathematical model for sampling by using the Dirac delta function δ . Define a pulse stream by

$$\forall t \in \mathbb{R}, \quad p(t) = \sum_{k=-\infty}^{\infty} \delta(t - kT).$$

Consider a continuous-time signal x that we wish to sample with sampling period T . That is, we define a discrete-time signal $y: \mathbb{Z} \rightarrow \mathbb{R}$ by $y(n) = x(nT)$. Construct first an intermediate continuous-time signal $w(t) = x(t)p(t)$. We can show that the Fourier transform of w is equal to the DTFT of y . This gives us a way to relate the Fourier transform of x to the DTFT of its samples y .

Recall that multiplication in the time domain results in convolution in the frequency domain, so

$$W(\omega) = \frac{1}{2\pi} X(\omega) * P(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\Omega) P(\omega - \Omega) d\Omega.$$

It can be shown (see box on page 126) that the Fourier transform of $p(t)$ is

$$P(\omega) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta(\omega - k\frac{2\pi}{T}),$$

so

$$\begin{aligned} W(\omega) &= \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\Omega) \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta(\omega - \Omega - k\frac{2\pi}{T}) d\Omega \\ &= \frac{1}{T} \sum_{k=-\infty}^{\infty} \int_{-\infty}^{\infty} X(\Omega) \delta(\omega - \Omega - k\frac{2\pi}{T}) d\Omega \\ &= \frac{1}{T} \sum_{k=-\infty}^{\infty} X(\omega - k\frac{2\pi}{T}) \end{aligned}$$

²This section requires a background in signals and systems at the level typically covered in a sophomore or junior engineering course.

where the last equality follows from the sifting property of Dirac delta functions. The next step is to show that

$$Y(\omega) = W(\omega/T),$$

which follows easily from the definition of the DTFT Y and the Fourier transform W . From this, the **Nyquist-Shannon sampling theorem** follows,

$$Y(\omega) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X\left(\frac{\omega - 2\pi k}{T}\right).$$

This relates the Fourier transform X of the signal being sampled x to the DTFT Y of the discrete-time result y .

This important relation says that the DTFT Y of y is the sum of the Fourier transform X with copies of it shifted by multiples of $2\pi/T$. Also, the frequency axis is normalized by dividing ω by T . There are two cases to consider, depending on whether the shifted copies overlap.

Probing Further: Impulse Trains

Consider a signal p consisting of periodically repeated Dirac delta functions with period T ,

$$\forall t \in \mathbb{R}, \quad p(t) = \sum_{k=-\infty}^{\infty} \delta(t - kT).$$

This signal has the Fourier series expansion

$$\forall t \in \mathbb{R}, \quad p(t) = \sum_{m=-\infty}^{\infty} \frac{1}{T} e^{i\omega_0 m t},$$

where the fundamental frequency is $\omega_0 = 2\pi/T$. The Fourier series coefficients can be given by

$$\forall m \in \mathbb{Z}, \quad P_m = \frac{1}{T} \int_{-T/2}^{T/2} \left[\sum_{k=-\infty}^{\infty} \delta(t - kT) \right] e^{i\omega_0 m t} dt.$$

The integral is over a range that includes only one of the delta functions. The kernel of the integral is zero except when $t = 0$, so by the sifting rule of the Dirac delta function, the integral evaluates to 1. Thus, all Fourier series coefficients are $P_m = 1/T$. Using the relationship between the Fourier series and the Fourier Transform of a periodic signal, we can write the continuous-time Fourier transform of p as

$$\forall \omega \in \mathbb{R}, \quad P(\omega) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi}{T}k\right).$$

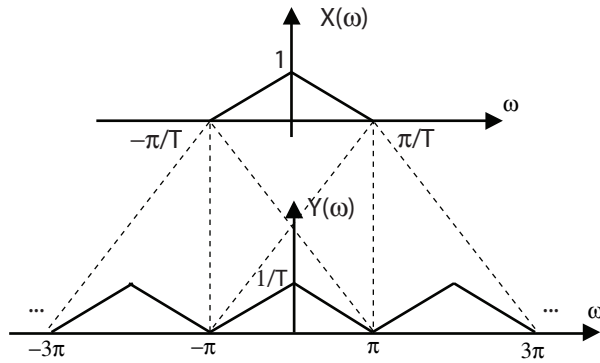


Figure 8.6: Relationship between the Fourier transform of a continuous-time signal and the DTFT of its discrete-time samples. The DTFT is the sum of the Fourier transform and its copies shifted by multiples of $2\pi/T$, the sampling frequency in radians per second. The frequency axis is also normalized.

First, if $X(\omega) = 0$ outside the range $-\pi/T < \omega < \pi/T$, then the copies will not overlap, and in the range $-\pi < \omega < \pi$,

$$Y(\omega) = \frac{1}{T} X\left(\frac{\omega}{T}\right). \quad (8.5)$$

In this range of frequencies, Y has the same shape as X , scaled by $1/T$. This relationship between X and Y is illustrated in figure 8.6, where X is drawn with a triangular shape.

In the second case, illustrated in figure 8.7, X does have non-zero frequency components higher than π/T . Notice that in the sampled signal, the frequencies in the vicinity of π are distorted by the overlapping of frequency components above and below π/T in the original signal. This distortion is called **aliasing distortion**.

From these figures, we get the guideline that we should sample continuous time signals at rates at least twice as high as the largest frequency component. This avoids aliasing distortion.

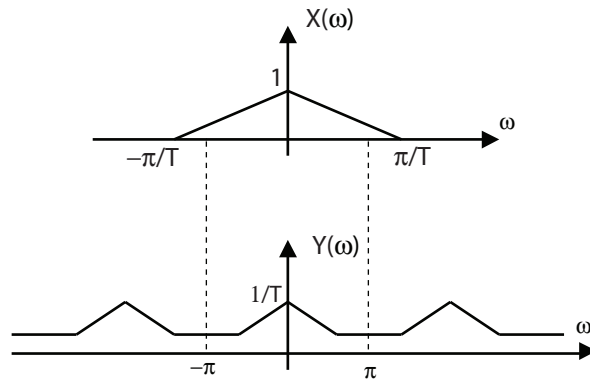


Figure 8.7: Relationship between the Fourier transform of a continuous-time signal and the DTFT of its discrete-time samples when the continuous-time signal has a broad enough bandwidth to introduce aliasing distortion.

Exercises

1. Figure 8.8 gives the sketch of a program for an Atmel AVR microcontroller that performs some function repeatedly for three seconds. The function is invoked by calling the procedure `foo()`. The program begins by setting up a timer interrupt to occur once per second (the code to do this setup is not shown). Each time the interrupt occurs, the specified interrupt service routine is called. That routine decrements a counter until the counter reaches zero. The `main()` procedure initializes the counter with value 3 and then invokes `foo()` until the counter reaches zero.
 - (a) We wish to assume that the segments of code in the grey boxes, labeled **A**, **B**, and **C**, are atomic. State conditions that make this assumption valid.
 - (b) Construct a state machine model for this program, assuming as in part (a) that **A**, **B**, and **C**, are atomic. The transitions in your state machine should be labeled with “guard/action”, where the action can be any of **A**, **B**, **C**, or nothing. The actions **A**, **B**, or **C** should correspond to the sections of code in the grey boxes with the corresponding labels. You may assume these actions are atomic.
 - (c) Is your state machine deterministic? What does it tell you about how many times `foo()` may be invoked? Do all the possible behaviors of your model correspond to what the programmer likely intended?

Note that there are many possible answers. Simple models are preferred over elaborate ones, and complete ones (where everything is defined) over incomplete ones. Feel free to give more than one model.

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;

// Interrupt service routine.
SIGNAL(SIG_OUTPUT_COMPARE1A) {

    if(timer_count > 0) {
        timer_count--;
    }

}

// Main program.
int main(void) {
    // Set up interrupts to occur
    // once per second.
    ...

    // Start a 3 second timer.
    timer_count = 3;

    // Do something repeatedly
    // for 3 seconds.
    while(timer_count > 0) {
        foo();
    }
}
```

Figure 8.8: Sketch of a C program that performs some function by calling procedure `foo()` repeatedly for 3 seconds, using a timer interrupt to determine when to stop.

2. In a manner similar to example 8.7, create a C program for the ARM CortexTM- M3 to use the SysTick timer to invoke a system-clock ISR with a jiffy interval of 10 ms that records the time since system start in a 32 bit int. How long can this program run before your clock overflows?
3. Suppose a processor handles interrupts exactly as specified by the FSM in Figure 8.5. Answer the following questions.
 - (a) If the state is *Pending* and the input is $\{return\}$ (meaning that only the *return* input symbol is present), what is the reaction?
 - (b) If the state is *Active* and the input is $\{assert, deassert\}$, what is the reaction?
 - (c) Suppose the state is *Inactive* and the input sequence is
 - i. $\{assert\}$,
 - ii. $\{deassert, handle\}$,
 - iii. $\{return\}$.What are all the possible states after reacting to these inputs? Was the interrupt handled or not?
 - (d) Suppose that an input sequence never includes *dessert*. Is it true that every *assert* input causes an *acknowledge* output? In other words, is every interrupt request serviced? If yes, give a proof. If no, give a counterexample.
4. Suppose you are designing a processor that will support two interrupts whose logic is given in Figure 8.5. Design an FSM giving the logic of an arbiter that assigns one of these two higher priority than the other. The input symbols should be

$$\{request1, return1, request2, return2\}$$

to indicate requests and return from interrupt for interrupts 1 and 2, respectively. The output symbols should be $\{handle1, handle2\}$. Assuming the request inputs are generated by two state machines like that in Figure 8.5, can you be sure that this arbiter will handle every request that is made? Justify your answer.

5. Consider the accelerometer problem described in Example 8.13. Suppose that the change in orientation x_d is a low frequency signal with Fourier transform given by

$$X_d(\omega) = \begin{cases} 2 & \text{for } |\omega| < \pi \\ 0 & \text{otherwise} \end{cases}$$

This is an ideally bandlimited signal with no frequency content higher than π radians/second, or 0.5 Hertz. Suppose further that the vibration x_n has higher frequency components, having Fourier transform given by

$$X_n(\omega) = \begin{cases} 1 & \text{for } |\omega| < 10\pi \\ 0 & \text{otherwise} \end{cases}$$

This is again an ideally bandlimited signal with frequency content up to 5 Hertz.

- (a) Assume there is no frequency conditioning at all, or equivalently, the conditioning filter has transfer function

$$\forall \omega \in \mathbb{R}, \quad H(\omega) = 1.$$

Find the SNR in decibels.

- (b) Assume the conditioning filter is an ideal lowpass filter with transfer function

$$H(\omega) = \begin{cases} 1 & \text{for } |\omega| < \pi \\ 0 & \text{otherwise} \end{cases}$$

Find the SNR in decibels. Is this better or worse than the result in part (a)? By how much?

- (c) Find a conditioning filter that makes the error signal identically zero (or equivalently makes the SNR infinite). Clearly, this conditioning filter is optimal for these signals. Explain why this isn't necessarily the optimal filter in general.
- (d) Suppose that as in part (a), there is no signal conditioning. Sample the signal x at 1 Hz and find the SNR of the resulting discrete-time signal.
- (e) Describe a strategy that minimizes the amount of signal conditioning that is done in continuous time in favor of doing signal conditioning in discrete time. The motivation for doing this is analog circuitry can be much more expensive than digital filters.
6. Consider the accelerometer problem described in Example 8.13. Suppose that the change in orientation x_d is a low frequency signal with Fourier transform given by

$$X_d(\omega) = \begin{cases} 2 & \text{for } |\omega| < \pi \\ 0 & \text{otherwise} \end{cases}$$

This is an ideally bandlimited signal with no frequency content higher than π radians/second, or 0.5 Hertz. Suppose further that the vibration x_n has higher frequency components, having Fourier transform given by

$$X_n(\omega) = \begin{cases} 1 & \text{for } |\omega| < 10\pi \\ 0 & \text{otherwise} \end{cases}$$

This is again an ideally bandlimited signal with frequency content up to 5 Hertz.

- (a) Assume there is no frequency conditioning at all, or equivalently, the conditioning filter has transfer function

$$\forall \omega \in \mathbb{R}, \quad H(\omega) = 1.$$

Find the SNR in decibels.

- (b) Assume the conditioning filter is an ideal lowpass filter with transfer function

$$H(\omega) = \begin{cases} 1 & \text{for } |\omega| < \pi \\ 0 & \text{otherwise} \end{cases}$$

Find the SNR in decibels. Is this better or worse than the result in part (a)? By how much?

EXERCISES

- (c) Find a conditioning filter that makes the error signal identically zero (or equivalently makes the SNR infinite). Clearly, this conditioning filter is optimal for these signals. Explain why this isn't necessarily the optimal filter in general.
- (d) Suppose that as in part (a), there is no signal conditioning. Sample the signal x at 1 Hz and find the SNR of the resulting discrete-time signal.
- (e) Describe a strategy that minimizes the amount of signal conditioning that is done in continuous time in favor of doing signal conditioning in discrete time. The motivation for doing this is analog circuitry can be much more expensive than digital filters.

Chapter 9

Multitasking

Contents

9.1 Processes	133
9.2 Threads	133
9.3 Message Passing	133

This chapter, when written, will discuss **processes** and **threads**, including various related techniques like **message passing**, **shared memory**, **mutual exclusion**, and **semaphores**, while defining terms like **lock**, **(mutex)**, **API (Application Program Interface)**, **bare iron**, **system clocks**, **jiffy**, and **address translation**. The chapter will draw on examples from video gaming (using **NTSC**, **PAL**, or **ATSC** television standards). It will also use **PThreads** and **Embedded Linux** to illustrate concepts. [\(p. 78\)](#)

9.1 Processes

9.2 Threads

9.3 Message Passing

Chapter 10

Networking and Distributed Systems

This chapter, yet to be written, gives an overview of networking technology.

Probing Further: Zigbee

10.1 ISO Layers

10.2 Physical and Link Layer

10.3 Ethernet

10.4 CAN

10.5 FlexRay

10.6 Wireless

10.7 Time Synchronization

10.7.1 NTP

10.7.2 IEEE 1588

Exercises

- 1.

Chapter 11

Scheduling

This chapter, when written, will discuss **real-time programs**, studying concepts such as **scheduling**, **schedulers**, **multiprocessor schedulers**, **processor assignment**, **precedence constraints**, **timing constraints**, including **deadlines**

Applications include **safety-critical systems**,

Terms that are defined include **preconditions**, **enabled**, **blocked**, **preemption**, **preemptive scheduler**, **non-preemptive scheduler**, **scheduling decision**, **release time**, **start time**, **finish time**, **response time**, **execution time** **design time** vs. **run time**, **fully static scheduler**, **static order scheduler**, **static assignment scheduler**, **run-time scheduler**, and **fully dynamic scheduler**.

Part III

Analysis and Verification

Chapter 12

Simulation

This chapter will introduce basic concepts for simulating models of discrete and continuous systems.

12.1 Efficient Random Sampling

12.2 Discrete Event Simulation

12.3 Numerical Simulation

12.4 Hardware-in-the-Loop Simulation

Chapter 13

Invariants and Temporal Logic

This chapter will introduce the notion of an invariant, and then generalize it to more expressive specifications in temporal logic.

13.1 Invariants

13.2 Safety and Liveness

13.3 Linear Temporal Logic

13.4 Other Temporal Logics

Chapter 14

Equivalence and Refinement¹

This chapter will discuss some fundamental notions of correctness for state machines and other modal models, such as trace equivalence, trace containment, simulation, and bisimulation.

14.1 Models as Specifications

14.2 Trace Equivalence and Containment

14.3 Simulation Relations

14.3.1 Relating behaviors

¹This chapter currently borrows heavily from [Lee and Varaiya \(2003\)](#).

Chapter 15

Reachability Analysis and Model Checking

This chapter will introduce the notion of the reachable set of states, relate it to system invariants, and discuss basic algorithms for reachability analysis and model checking.

15.1 Reachability and Invariants

15.2 Reachability Analysis

15.3 Model Checking

Chapter 16

Quantitative Analysis

This chapter will discuss the analysis of quantitative properties of embedded systems, such as execution time, memory, and power/energy consumption.

16.1 The Need for Quantitative Analysis

16.2 Execution Time Analysis

16.3 Memory Analysis

16.4 Power Analysis

Part IV

Appendices

Appendix A

Sets and Functions

This appendix reviews some basic notation on sets and functions.

A.1 Sets

In this section, we review the notation for sets. A **set** is a collection of objects. When object a is in set A , we write $a \in A$. We define the following sets:

- $\mathbb{B} = \{0, 1\}$, the set of **binary digits**.
- $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of **natural numbers**.
- $\mathbb{Z} = \{\dots, -1, 0, 1, 2, \dots\}$, the set of **integers**.
- $\mathbb{R}$, the set of **real numbers**.
- $\mathbb{R}_+$, the set of **non-negative real numbers**.

When set A is entirely contained by set B , we say that A is a **subset** of B and write $A \subseteq B$. For example, $\mathbb{B} \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$. The sets may be equal, so the statement $\mathbb{N} \subseteq \mathbb{N}$ is true, for example. The **powerset** of a set A is defined to be the set of all subsets. It is written 2^A . The **empty set**, written $\emptyset$, is always a member of the powerset, $\emptyset \in 2^A$.

We define **set subtraction** as follows,

$$A \setminus B = \{a \in A : a \notin B\}$$

for all sets A and B . This notation is read “the set of elements a from A such that a is not in B .”

A **cartesian product** of sets A and B is a set written $A \times B$ and defined as follows,

$$A \times B = \{(a, b) : a \in A, b \in B\}.$$

A member of this set (a, b) is called a **tuple**. This notation is read “the set of tuples (a, b) such that a is in A and b is in B .” A cartesian product can be formed with three or more sets, in which case the tuples have three or more elements. For example, we might write $(a, b, c) \in A \times B \times C$. A cartesian product of a set A with itself is written $A^2 = A \times A$. A cartesian product of a set A with itself n times, where $n \in \mathbb{N}$ is written A^n . A member of the set A^n is called an **n -tuple**. By convention, A^0 is a **singleton set**, or a set with exactly one element, regardless of the size of A . Specifically, we define $A^0 = \{\emptyset\}$. Note that A^0 is not itself the empty set. It is a singleton set containing the empty set (for insight into the rationale for this definition, see the box on page 158).

A.2 Relations and Functions

A **relation** from set A to set B is a subset of $A \times B$. A **partial function** f from set A to set B is a relation where $(a, b) \in f$ and $(a, b') \in f$ imply that $b = b'$. Such a partial function is written $f: A \rightharpoonup B$. A **total function** or just **function** f from A to B is a partial function where for all $a \in A$, there is a $b \in B$ such that $(a, b) \in f$. Such a function is written $f: A \rightarrow B$, and the set A is called its **domain** and the set B its **codomain**. Rather than writing $(a, b) \in f$, we can equivalently write $f(a) = b$.

Example 1.1: An example of a partial function is $f: \mathbb{R} \rightharpoonup \mathbb{R}$ defined by $f(x) = \sqrt{x}$ for all $x \in \mathbb{R}_+$. It is undefined for any $x < 0$ in its domain $\mathbb{R}$.

A partial function $f: A \rightharpoonup B$ may be defined by an **assignment rule**, as done in the above example, where an assignment rule simply explains how to obtain the value of $f(a)$ given $a \in A$. Alternatively, the function may be defined by its **graph**, which is a subset of $A \times B$.

Example 1.2: The same partial function from the previous example has the graph $f \subseteq \mathbb{R}^2$ given by

$$f = \{(x, y) \in \mathbb{R}^2 : x \geq 0 \text{ and } y = \sqrt{x}\}.$$

Note that we use the same notation f for the function and its graph when it is clear from context which we are talking about.

The **set of all functions** $f: A \rightarrow B$ is written $(A \rightarrow B)$ or B^A . The former notation is used when the exponential notation proves awkward. For a justification of the notation B^A , see the box on page 158.

The **function composition** of $f: A \rightarrow B$ and $g: B \rightarrow C$ is written $(g \circ f): A \rightarrow C$ and defined by

$$(g \circ f)(a) = g(f(a))$$

for any $a \in A$. Note that in the notation $(g \circ f)$, the function f is applied first. For a function $f: A \rightarrow A$, the composition with itself can be written $(f \circ f) = f^2$, or more generally

$$\underbrace{(f \circ f \circ \dots \circ f)}_{n \text{ times}} = f^n$$

for any $n \in \mathbb{N}$. In case $n = 1$, $f^1 = f$. For the special case $n = 0$, the function f^0 is by convention the **identity function**, so $f^0(a) = a$ for all $a \in A$. When the domain and codomain of a function are the same, i.e. $f \in A^A$, then $f^n \in A^A$ for all $n \in \mathbb{N}$.

For every function $f: A \rightarrow B$, there is an associated **image function** $\hat{f}: 2^A \rightarrow 2^B$ defined on the powerset of A as follows,

(p. 153)

$$\forall A' \subseteq A, \quad \hat{f}(A') = \{b \in B : \exists a \in A', f(a) = b\}.$$

We call $\hat{f}$ the **lifted** version of f . When there is no ambiguity, we may write the lifted version of f simply as f rather than $\hat{f}$ (see problem 2(c) for an example of a situation where there is ambiguity).

For any $A' \subseteq A$, $\hat{f}(A')$ is called the **image** of A' for the function f . The image $\hat{f}(A)$ of the domain is called the **range** of the function f .

Example 1.3: The image $\hat{f}(\mathbb{R})$ of the function $f: \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x) = x^2$ is $\mathbb{R}_+$.

A function $f: A \rightarrow B$ is **onto** (or **surjective**) if $\hat{f}(A) = B$. A function $f: A \rightarrow B$ is **one-to-one** (or **injective**) if for all $a, a' \in A$,

$$a \neq a' \Rightarrow f(a) \neq f(a'). \quad (\text{A.1})$$

That is, no two distinct values in the domain yield the same values in the codomain. A function that is both one-to-one and onto is called a **bijection**.

Example 1.4: The function $f: \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x) = 2x$ is a bijection. The function $f: \mathbb{Z} \rightarrow \mathbb{Z}$ defined by $f(x) = 2x$ is one-to-one, but not onto. The function $f: \mathbb{R}^2 \rightarrow \mathbb{R}$ defined by $f(x, y) = xy$ is onto but not one-to-one.

The previous example underscores the fact that an essential part of the definition of a function is its domain and codomain.

Proposition A.1. *If $f: A \rightarrow B$ is onto, then there is a one-to-one function $h: B \rightarrow A$.*

Proof. Let h be defined by $h(b) = a$ where a is any element in A such that $f(a) = b$. There must always be at least one such element because f is onto. We can now show that h is one-to-one. To do this, consider any two elements $b, b' \in B$ where $b \neq b'$. We need to show that $h(b) \neq h(b')$. Assume to the contrary that $h(b) = h(b') = a$ for some $a \in A$. But then by the definition of h , $f(a) = b$ and $f(a) = b'$, which implies $b = b'$, a contradiction. $\square$

The converse of this proposition is also easy to prove.

Proposition A.2. *If $h: B \rightarrow A$ is one-to-one, then there is an onto function $f: A \rightarrow B$.*

Any bijection $f: A \rightarrow B$ has an **inverse** $f^{-1}: B \rightarrow A$ defined as follows,

$$f^{-1}(b) = a \in A \text{ such that } f(a) = b, \quad (\text{A.2})$$

for all $b \in B$. This function is defined for all $b \in B$ because f is onto. And for each $b \in B$ there is a single unique $a \in A$ satisfying (A.2) because f is one-to-one. For any bijection f , its inverse is also a bijection.

A.2.1 Restriction and Projection

Given a function $f: A \rightarrow B$ and a subset $C \subseteq A$, we can define a new function $f|_C$ that is the **restriction** of f to C . It is defined so that for all $x \in C$, $f|_C(x) = f(x)$.

Example 1.5: The function $f: \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x) = x^2$ is not one-to-one. But the function $f|_{\mathbb{R}_+}$ is.

Consider an n -tuple $a = (a_0, a_1, \dots, a_{n-1}) \in A_0 \times A_1 \times \dots \times A_{n-1}$. A **projection** of this n -tuple extracts elements of the tuple to create a new tuple. Specifically, let

$$I = (i_0, i_1, \dots, i_m) \in \{0, 1, \dots, n-1\}^m$$

for some $m \in \mathbb{N} \setminus \{0\}$. That is, I is an m -tuple of indexes. Then we define the projection of a onto I by

$$\pi_I(a) = (a_{i_0}, a_{i_1}, \dots, a_{i_m}) \in A_{i_0} \times A_{i_1} \times \dots \times A_{i_m}.$$

The projection may be used to permute elements of a tuple, to discard elements, or to repeat elements.

Projection of a tuple and restriction of a function are related. An n -tuple $a \in A^n$ where $a = (a_0, a_1, \dots, a_{n-1})$ may be considered a function of the form $a: \{0, 1, \dots, n-1\} \rightarrow A$, in which case $a(0) = a_0, a(1) = a_1$, etc. Projection is similar to restriction of this function, differing in that restriction, by itself, does not provide the ability to permute, repeat, or renumber elements. But conceptually, the operations are similar, as illustrated by the following example.

Example 1.6: Consider a 3-tuple $a = (a_0, a_1, a_2) \in A^3$. This is represented by the function $a: \{0, 1, 2\} \rightarrow A$. Let $I = \{1, 2\}$. The projection $b = \pi_I(a) = (a_1, a_2)$, which itself can be represented by a function $b: \{0, 1\} \rightarrow A$, where $b(0) = a_1$ and $b(1) = a_2$.

The restriction $a|_I$ is not exactly the same function as b , however. The domain of the first function is $\{1, 2\}$, whereas the domain of the second is $\{0, 1\}$. In particular, $a|_I(1) = b(0) = a_1$ and $a|_I(2) = b(1) = a_2$.

A projection may **lifted** just like ordinary functions. Given a set of n -tuples $B \subseteq A_0 \times A_1 \times \dots \times A_{n-1}$ (p. 155) and an m -tuple of indexes $I \in \{0, 1, \dots, n-1\}^m$, the **lifted projection** is

$$\hat{\pi}_I(B) = \{\pi_I(b) : b \in B\}.$$

A.3 Sequences

A tuple $(a_0, a_1) \in A^2$ can be interpreted as a sequence of length 2. The order of elements in the sequence matters, and is in fact captured by the natural ordering of the natural numbers. The number 0 comes before the number 1. We can generalize this and recognize that a **sequence** of elements from set A of length n is an n -tuple in the set A^n . A^0 represents the set of empty sequences, a **singleton set** (there is only one empty sequence). (p. 154)

The set of all **finite sequences** of elements from the set A is written A^* , where we interpret $*$ as a wildcard that can take on any value in $\mathbb{N}$. A member of this set with length n is an n -tuple.

The set of **infinite sequences** of elements from A is written $A^{\mathbb{N}}$ or A^{ω} . The set of **finite and infinite sequences** is written

$$A^{**} = A^* \cup A^{\mathbb{N}}.$$

Finite and infinite sequences play an important role in the semantics of concurrent programs. They can be used, for example, to represent streams of messages sent from one part of the program to another. Or they can represent successive assignments of values to a variable. For programs that terminate, finite sequences will be sufficient. For programs that do not terminate, we need infinite sequences.

Exercises

1. This problem explores properties of onto and one-to-one functions.
 - (a) Show that if $f: A \rightarrow B$ is onto and $g: B \rightarrow C$ is onto, then $(g \circ f): A \rightarrow C$ is onto.
 - (b) Show that if $f: A \rightarrow B$ is one-to-one and $g: B \rightarrow C$ is one-to-one, then $(g \circ f): A \rightarrow C$ is one-to-one.

Insight: Exponential Notation for Sets of Functions

(p. 153)

The exponential notation B^A for the set of functions of form $f: A \rightarrow B$ is worth explaining. Recall that A^2 is the **cartesian product** of set A with itself, and that 2^A is the **powerset** of A . These two notations are naturally thought of as sets of functions. A construction attributed to John von Neumann defines the natural numbers as follows,

$$\begin{aligned}
 \mathbf{0} &= \emptyset \\
 \mathbf{1} &= \{\mathbf{0}\} = \{\emptyset\} \\
 \mathbf{2} &= \{\mathbf{0}, \mathbf{1}\} = \{\emptyset, \{\emptyset\}\} \\
 \mathbf{3} &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\
 &\dots
 \end{aligned}$$

With this definition, the powerset 2^A is the set of functions mapping the set A into the set $\mathbf{2}$. Consider one such function, $f \in 2^A$. For each $a \in A$, either $f(a) = \mathbf{0}$ or $f(a) = \mathbf{1}$. If we interpret “ $\mathbf{0}$ ” to mean “nonmember” and “ $\mathbf{1}$ ” to mean “member,” then indeed the set of functions 2^A represents the set of all subsets of A . Each such function defines a subset.

Similarly, the cartesian product A^2 can be interpreted as the set of functions of form $f: \mathbf{2} \rightarrow A$, or using von Neumann’s numbers, $f: \{\mathbf{0}, \mathbf{1}\} \rightarrow A$. Consider a tuple $a = (a_0, a_1) \in A^2$. It is natural to associate with this tuple a function $a: \{\mathbf{0}, \mathbf{1}\} \rightarrow A$ where $a(\mathbf{0}) = a_0$ and $a(\mathbf{1}) = a_1$. The argument to the function is the index into the tuple. We can now interpret the set of functions B^A of form $f: A \rightarrow B$ as a set of tuples indexed by the set A instead of by the natural numbers.

Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \dots\}$ represent the set of **von Neumann numbers**. This set is closely related to the set $\mathbb{N}$ (see problem 2). Given a set A , it is now natural to interpret A^ω as the set of all infinite sequences of elements from A , the same as $A^\mathbb{N}$.

(p. 154)

The **singleton set** $A^{\mathbf{0}}$ can now be interpreted as the set of all functions whose domain is the empty set and codomain is A . There is exactly one such function (no two such functions are distinguishable), and that function has an empty **graph**. Before, we defined $A^{\mathbf{0}} = \{\emptyset\}$. Using von Neumann numbers, $A^{\mathbf{0}} = \mathbf{1}$, corresponding nicely with the definition of a zero exponent on ordinary numbers. Moreover, you can think of $A^{\mathbf{0}} = \{\emptyset\}$ as the set of all functions with an empty graph.

(p. 154)

It is customary in the literature to omit the bold face font for $A^{\mathbf{0}}$, 2^A , and A^2 , writing instead simply A^0 , 2^A , and A^2 .

2. Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \dots\}$ be the [von Neumann numbers](#) as defined in the box on page [158](#). This problem explores the relationship between this set and $\mathbb{N}$, the set of natural numbers. (p. 158)

(a) Let $f: \omega \rightarrow \mathbb{N}$ be defined by

$$f(x) = |x|, \quad \forall x \in \omega.$$

That is, $f(x)$ is the size of the set x . Show that f is a [bijection](#). (p. 155)

- (b) The [lifted](#) version of the function f in part (a) is written $\hat{f}$. What is the value of $\hat{f}(\{0, \{0\}\})$? What is the value of $f(\{0, \{0\}\})$? Note that on page [155](#) it is noted that when there is no ambiguity, $\hat{f}$ may be written simply f . For this function, is there such ambiguity? (p. 155)

EXERCISES

Bibliography

- Abelson, H. and G. J. Sussman, 1996: *Structure and Interpretation of Computer Programs*. MIT Press, 2nd ed.
- Allen, J. ., 1975: Computer architecture for signal processing. *Proceedings of the IEEE*, **63(4)**, 624–633.
- André, C., 1996: SyncCharts: a visual representation of reactive behaviors. Tech. Rep. RR 95–52, rev. RR (96–56), University of Sophia-Antipolis.
- ARM Limited, 2006: CortexTM- M3 technical reference manual. Tech. rep.,
?http://www.arm.com.
- Barr, M. and A. Massa, 2006: *Programming Embedded Systems*. O'Reilly, second edition ed.
- Beeck, M. v. d., 1994: A comparison of Statecharts variants. In Langmaack, H., W. P. de Roever, and J. Vytöpil, eds., *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, Lübeck, Germany, vol. 863 of *Lecture Notes in Computer Science*, pp. 128–148.
- Berger, A. S., 2002: *Embedded Systems Design: An Introduction to Processes, Tools, & Techniques*. CMP Books.
- Berry, G. and G. Gonthier, 1992: The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, **19(2)**, 87–152.
- Booch, G., I. Jacobson, and J. Rumbaugh, 1998: *The Unified Modeling Language User Guide*. Addison-Wesley.
- Burns, A. and A. Wellings, 2001: *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 3rd ed.
- Buttazzo, G. C., 2005: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2nd ed.
- Derenzo, S. E., 2003: *Practical Interfacing in the Laboratory: Using a PC for Instrumentation, Data Analysis and Control*. Cambridge University Press.

BIBLIOGRAPHY

- Eden, M. and M. Kagan, 1997: The Pentium®processor with MMX™technology. In *IEEE International Conference (COMPCON)*, IEEE, San Jose, CA, USA, pp. 260–262.
- Edwards, S. A., 2000: *Languages for Digital Embedded Systems*. Kluwer Academic Publishers.
- Gajski, D. D., S. Abdi, A. Gerstlauer, and G. Schirner, 2009: *Embedded System Design - Modeling, Synthesis, and Verification*. Springer.
- Goldstein, H., 1980: *Classical Mechanics*. Addison-Wesley, 2nd ed.
- Harel, D., 1987: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8**, 231–274.
- Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, **16(4)**.
- Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.
- Jantsch, A., 2003: *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.
- Kamal, R., 2008: *Embedded Systems: Architecture, Programming, and Design*. McGraw Hill.
- Landau, L. D. and E. M. Lifshitz, 1976: *Mechanics*. Pergamon Press, 3rd ed.
- Lapsley, P., J. Bier, A. Shoham, and E. A. Lee, 1997: *DSP Processor Fundamentals – Architectures and Features*. IEEE Press, New York.
- Lee, E. A., 2009: Computing needs time. Tech. Rep. UCB/EECS-2009-30, EECS Department, University of California, Berkeley.
- Lee, E. A. and P. Varaiya, 2003: *Structure and Interpretation of Signals and Systems*. Addison Wesley.
- Lee, R. B., 1996: Subword parallelism with MAX2. *IEEE Micro*, **16(4)**, 51–59.
- Liu, J. W. S., 2000: *Real-Time Systems*. Prentice-Hall.
- Luminary Micro®, 2008a: Stellaris® LM3S8962 evaluation board user’s manual. Tech. rep., Luminary Micro, Inc.
- , 2008b: Stellaris® LM3S8962 microcontroller data sheet. Tech. rep., Luminary Micro, Inc.
- , 2008c: Stellaris® peripheral driver library - user’s guide. Tech. rep., Luminary Micro, Inc.
- Marion, J. B. and S. Thornton, 1995: *Classical Dynamics of Systems and Particles*. Thomson, 4th ed.

- Marwedel, P., 2003: *Embedded System Design*. Kluwer Academic Publishers.
- Mishra, P. and N. D. Dutt, 2005: *Functional Verification of Programmable Embedded Processors - A Top-down Approach*. Springer.
- Noergaard, T., 2005: *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Elsevier.
- Oshana, R., 2006: *DSP Software Development Techniques for Embedded and Real-Time Systems*. Embedded Technology Series, Elsevier.
- Parab, J. S., V. G. Shelake, R. K. Kamat, and G. M. Naik, 2007: *Exploring C for Microcontrollers*. Springer.
- Patterson, D. A. and D. R. Ditzel, 1980: The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, **8(6)**, 25–33.
- Patterson, D. A. and J. L. Hennessey, 1996: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition ed.
- Pottie, G. and W. Kaiser, 2005: *Principles of Embedded Networked Systems Design*. Cambridge University Press.
- Simon, D. E., 2006: *An Embedded Software Primer*. Addison-Wesley.
- Sriram, S. and S. S. Bhattacharyya, 2009: *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, second edition ed.
- Stankovic, J. A., I. Lee, A. Mok, and R. Rajkumar, 2005: Opportunities and obligations for physical computing systems. *Computer*, 23–31.
- Tremblay, M., J. M. O'Connor, V. Narayannan, and H. Liang, 1996: VIS speeds new media processing. *IEEE Micro*, **16(4)**, 10–20.
- Valvano, J. W., 2007: *Embedded Microcomputer Systems - Real Time Interfacing*. Thomson, second edition ed.
- Wolf, W., 2000: *Computers as Components: Principles of Embedded Computer Systems Design*. Morgan Kaufman.

Notation Index

$\neg$	negation	27
$\wedge$	conjunction	27
$\vee$	disjunction	27
$:=$	assignment	48
V_{CC}	supply voltage	106
$\mathbb{B} = \{0, 1\}$	binary digits	153
$\mathbb{N} = \{0, 1, 2, \dots\}$	natural numbers	153
$\mathbb{Z} = \{\dots, -1, 0, 1, 2, \dots\}$	integers	153
$\mathbb{R}$	real numbers	153
$\mathbb{R}_+$	non-negative real numbers	153
$A \subseteq B$	subset	153
2^A	powerset	153
$\emptyset$	empty set	153
$A \setminus B$	set subtraction	153
$A \times B$	cartesian product	154
$(a, b) \in A \times B$	tuple	154
A^0	singleton set	154
$f: A \rightarrow B$	function	154
$f: A \rightharpoonup B$	partial function	154
$g \circ f$	function composition	155
$f^n: A \rightarrow A$	function to a power	155
$f^0(a)$	identity function	155
$\hat{f}: 2^A \rightarrow 2^B$	image function	155
$(A \rightarrow B)$	set of all functions from A to B	155
B^A	set of all functions from A to B	155
π_I	projection	156
$\hat{\pi}_I$	lifted projection	157
$f _C$	restriction	156
A^*	finite sequences	157
$A^{\mathbb{N}}$	infinite sequences	157
$\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \dots\}$	von Neumann numbers	158
A^ω	infinite sequences	158

NOTATION INDEX

A^{**}	finite and infinite sequences	157
----------	-------------------------------	-----

Index

- n*-tuple, [154](#)
- 1080p, [79](#)
- 32-bit architecture, [101](#)
- 32-bit microprocessor, [96](#)
- 3D graphics, [81](#)
- 64-bit microprocessor, [96](#)
- 8-bit microcontroller, [96](#)

- 6800, [75](#)
- 6811, [75](#)
- 8051, [75](#)
- 8080, [75](#)
- 8086, [76](#)
- 80386, [76](#)

- abstraction, [xi](#)
- acceleration, [5](#)
- acceleration of gravity, [54](#)
- accelerometer, [71](#)
- accumulator, [80](#)
- action, [27](#)
- active high logic, [106](#)
- active low logic, [106](#)
- actor, [9](#)
- actor model for state machines, [44](#)
- actor models, [8](#)
- adaptive antennas, [87](#)
- ADC, [121](#)
- adder, [10](#)
- address translation, [133](#)
- ADL, [xi](#)
- Advanced Television Systems Committee, [78](#)
- AGV, [56](#)
- Aiken, Howard H., [98](#)
- Alcatel, [75](#)
- aliasing distortion, [127](#)
- alignment, [101](#)
- allocation, [103](#)
- ALU, [90](#)
- AMD, [74](#), [76](#), [81](#)
- AMI, [76](#)
- amplifier, [107](#)
- analog, [105](#), [121](#)
- analog comparator, [121](#)
- analog to digital converter, [121](#)
- API, [133](#)
- Apple, [75](#)
- Application Program Interface, [133](#)
- architecture description languages, [xi](#)
- arithmetic logic unit, [90](#)
- ARM, [75](#), [98](#), [101](#), [118](#)
- ARM Cortex™- M3, [98](#), [100](#), [106](#), [114](#), [130](#)
- ARM instruction set, [116](#)
- ARM Limited, [75](#)
- ArrivalDetector, [22](#)
- asserting an interrupt, [118](#)
- assignment, [48](#)
- assignment rule, [154](#)
- asynchronous, [111](#)
- Atmel, [75](#), [128](#)
- Atmel AVR, [75](#)
- Atom, Intel, [74](#), [76](#)
- atomic operations, [116](#)
- ATSC, [78](#)
- audio, [76](#), [121](#)
- auto increment, [76](#)
- automata, [47](#)
- automated guided vehicle, [56](#)
- automatic variable, [104](#)

- automotive, 71, 77
- AVR, 75, 128
- bare iron, 133
- baseband processing, 76
- baud, 111
- behavior, 37
- Bell Labs, 76
- BIBO stable, 12
- big endian, 101
- bijection, 155
- binary digits, 153
- binary point, 88
- bipolar transistor, 109
- bit banding, 99
- bit reversed addressing, 76
- Blefuscu, 101
- blocked, 137
- Bogen, Alf-Egil, 75
- boundary scan, 112
- bounded, 12
- bounded-input bounded-output stable, 12
- breakpoint, 113
- brittleness, xii
- Broadcom, 75
- byte, 101
- C data type, 111
- C programming language, 81
- C54x, 84, 85
- C55x, 87
- c6000, 87
- c62x, 87
- C64x, 93
- c64x, 87
- c67x, 87
- cache, 99
- cache miss, 99
- cartesian product, 154
- cascade composition, 9
- CD, 97, 121
- CDs, 75
- cellular base stations, 87
- central processing unit, 74
- Centronics, 113
- char, 101
- chattering, 28
- chemical processes, 71
- chip, 74
- chrominance, 79
- circular buffer, 80
- circular buffers, 76
- Cirrus Logic, 75
- CISC, 84
- classical mechanics, 4
- clock, 47
- CMOS, 109
- codomain, 154
- ColdFire, 75
- color, 78, 85
- color channels, 79
- compact disc, 121
- compiler, 82
- complex instruction set computer, 84
- computation tree, 38
- computed branch, 90
- concurrent, x
- conditional branch, 89
- conditional branch instruction, 92
- conditioning filter, 123
- conjunction, 27
- consumer electronics, 87
- continuous, 21
- continuous state, 50
- continuous-time signal, 5
- control hazard, 92
- controller, 55
- convolution, 125
- core, 97
- Cortex, 98, 100, 106, 114, 130
- Cortex™, 98, 118
- Counter, 22
- CPS, ix
- CPU, 74
- crystal oscillators, 72

- current state, 32
- cyber-physical system, ix
- Cypress Semiconductor, 75
- Cyrix, 76
- DAC, 122
- Dallas Semiconductor, 75
- data hazard, 91
- dataflow, 67
- dataflow analysis, 82
- dB, 123
- DB-25, 113
- DE-9, 111
- deadline, 137
- deallocate, 103
- debugging, 113
- decibels, 123
- decode pipeline stage, 89
- default transition, 29
- defragmentation, 103
- delay, 12
- delay line, 77
- delayed branch, 92
- delta function, 125
- delta functions, 126
- DepartureDetector, 22
- design time, 137
- desired part, 122
- determinate, xi
- deterministic, 33
- device driver, 120
- differential equation, 4
- digital, 105
- digital signal, 121
- digital signal processor, 76, 77
- digital television, 87
- digital to analog converter, 122
- Dirac delta function, 125
- Dirac delta functions, 126
- discrete, 21, 23, 41
- discrete dynamics, 23
- discrete event, 22
- discrete signal, 25
- discrete state, 50
- discrete-time Fourier transform, 125
- disjunction, 27
- disk, 97
- disk drive controllers, 76
- disk drives, 98
- domain, 154
- double, 101
- DRAM, 96
- driver assist, 77
- DSP, 76, 77
- DSP1, 76
- DTFT, 125
- DVD, 97
- dynamic memory allocation, 103
- dynamic random-access memory, 96
- dynamics, xi
- edge triggered, 118
- EEPROM, 97
- EIA, 111
- electrical isolation, 108
- Electronics Industries Association, 111
- Embedded Linux, 133
- embedded software, ix
- embedded systems, ix
- empty set, 153
- enabled, 137
- encoder, 71
- energy, 123
- energy conservation, 28
- energy:of a discrete-time signal, 124
- engine controller, 83
- engine controllers, 71
- environment, 33, 55
- error signal, 13
- exception, 114
- execute pipeline stage, 89
- execution time, 137
- explicit pipeline, 91
- $f=ma$, 52
- fast Fourier transforms, 84

- fault handler, 23
- feature extraction, 77
- fetch pipeline stage, 89
- FFT, 76, 84
- fidelity, 18
- field-programmable gate array, 93
- filtering, 77
- finish time, 137
- finite and infinite sequences, 157
- finite impulse response, 77
- finite sequences, 157
- finite state machine, 46
- finite-state machine, 26
- FIR, 77, 78
 - two dimensional, 79
- firmware, 97
- fixed-point, 88
- flash memory, 97
- floating-point, 87
- floating-point standard, 90
- force, 5
- forwarding, 91
- Fourier transform, 124
- FPGA, 93
- fragmentation, 103
- frame, 76
- free method, 103
- Freescale, 75
- Freescale 6811, 75
- Freescale ColdFire, 75
- FSM, 26
- fully dynamic scheduler, 137
- fully static scheduler, 137
- function, 154
- function composition, 155
- function pointer, 115
- games, 77, 81
- garbage collection, 103
- gasoline engine, 83
- GB, 98
- general-purpose computing, x
- general-purpose I/O, 106
- Geode, 74
- GHz, 76
- gigabytes, 98
- Gill, Helen, ix
- GPIO, 113
- GPIO, 106
- GPU, 81
- graph, 154
- graphics, 76, 81
- graphics processing unit, 81
- gravity, 54
- grayscale, 79
- guard, 27
- Gullivers Travels, 101
- gyroscope, 71
- H8, 75
- hardware interrupt, 114
- Harvard architecture, 98
- heap, 103
- helicopter, 7
- Hennessy, John, 75
- Hertz, 75
- heterogeneous multicore, 92
- Hewlett Packard, 86
- Hewlett-Packard, 113
- hexadecimal notation, 101
- Hitachi H8, 75
- Hopper, Grace M., 98
- HP-IB, 113
- HVAC, 28
- hybrid system, 21, 43
- hybrid systems modeling, 4
- hysteresis, 28, 30
- Hz, 75
- I²C, 113
- I/O, 106
- IBM, 75, 98
- IBM PC, 76, 113
- icons, 10
- IDE, 106
- identity function, 155

- IEEE 1149.1, 112
- IEEE 754, 90, 101
- IEEE floating point standard, 101
- IEEE-1284, 113
- IEEE-488, 113
- ILP, 84
- image, 155
- image function, 155
- imperative, 81
- impulse response, 124
- incremental garbage collection, 104
- inductive coupling, 108
- industrial automation, 114
- inelastic, 54
- Infineon Technologies, 75
- infinite sequences, 157
- initial state, 26
- initially at rest, 14
- injective, 155
- instruction memory, 89
- instruction set architecture, 74, 96
- instruction-level parallelism, 84
- instrumentation, 77, 81
- int, 101
- integers, 153
- integral equation, 4
- integrated development environment, 106
- Intel, 81, 86
- Intel 80386, 76
- Intel 8051, 75
- Intel 8080, 75
- Intel 8086, 76
- Intel Atom, 74, 76
- Intel x86, 73
- intensity, 78
- inter-integrated circuit, 113
- interlock, 91
- interrupt, 114
- interrupt controller, 118, 119
- interrupt handler, 114
- interrupt service routine, 114
- interrupt vector, 118
- interrupt vector table, 118
- inverse, 156
- ISA, 74, 113
- ISR, 114
- Java, 103
- Java programming language, 81
- jiffy, 133
- joint test action group, 112
- Jonathan Swift, 101
- JTAG, 112
- kHz, 75
- kinetic energy, 54
- last-in, first-out, 102
- latch, 88
- law of conservation of bits, 89
- LCD display, 99
- LDM instruction, 116
- level triggered, 118
- LG, 75
- lidar, 77
- LIFO, 102
- lifted, 155
- lifted projection, 157
- Lilliput, 101
- linear phase, 85
- linear systems, 11
- linear time-invariant system, 12
- little endian, 101
- LM3S8962, 107
- LM3S8962 controller, 98, 118
- local variable, 104
- lock, 133
- logical addresses, 99
- low-level control, 56
- LTI, 12
- luminance, 79
- Luminary Micro, 98, 106, 107, 118
- machine learning, 77
- machine vision, 87
- magnetic core memory, 97

- malloc method, 103
- Mark I, 98
- Marvell Technology Group, 75
- mask ROM, 97
- Masuoka, Fujio, 97
- MathWorks, Inc., 4
- medical electronics, 77
- medical imaging, 75, 87
- memory addresses, 101
- memory fragmentation, 103
- memory hierarchy, 95, 97
- memory leak, 103
- memory management unit, 99
- memory map, 98, 100
- memory model, 101
- memory pipeline stage, 90
- memory protection, 103
- memory-mapped register, 99
- message passing, 133
- Microchip Technology, 75
- microcomputer board, 106
- microcontroller, 74, 111
- MicroSD, 99
- MIDI, 113
- MIPS, 75
- MMU, 99
- modal model, 47
- mode, 47, 50
- model, 3
- model-order reduction, 7
- modem, 111
- modems, 76
- moment of inertia tensor, 6
- momentum, 53
- motion control, 75
- Motorola, 75
- Motorola 6800, 75
- Motorola ColdFire, 75
- mph, 56
- multicore, 92
- multiply-accumulate instruction, 84
- multiprocessor scheduler, 137
- multitasking operating system, 83
- music, 75
- musical instrument digital interface, 113
- mutex, 133
- mutual exclusion, 133
- NAND flash, 97
- National Science Foundation, ix
- natural numbers, 153
- NEC, 75, 76
- negation, 27
- netbook, 74, 76
- neural impulse propagation, 108
- Newton's law, 51
- Newton's second law, 5
- next state, 32
- no-op, 91
- noise, 122
- non-negative real numbers, 153
- non-preemptive scheduler, 137
- non-volatile memory, 97
- nondeterministic FSM, 35
- NOR flash, 97
- Norwegian Institute of Technology, 75
- NTSC, 133
- NVIDIA, 75, 81
- NXP, 75, 87
- Nyquist-Shannon sampling theorem, 122, 126
- ODE, 4
- Ohm's law, 107
- OMAP, 92
- OMAP4440, 93
- one-to-one, 155
- onto, 155
- open collector, 109
- open drain, 109, 110
- optical media, 97
- opto isolator, 108
- order preserving, 25
- ordinary differential equation, 4
- out-of-order execution, 91
- overcurrent protection, 71

- overflow, 89
- PA RISC processor, 86
- page fault, 99
- PAL, 133
- parallel, 81
- parallel interface, 113
- Parallel ATA, 113
- parameter, 104
- Parseval's theorem, 124
- partial function, 154
- pass by reference, 104
- pass by value, 104
- pause time, 103
- PC, 89
- PCI, 113
- PCI Express, 113
- PDA, 84
- Pentium, 86
- performance, 82
- peripheral bus, 113
- peripheral component interconnect express, 113
- peripherals, 98
- permuting elements of a tuple, 156
- personal digital assistants, 84
- Philips, 87
- physical addresses, 99
- physical system, 3
- PIC, 75
- pipeline, 88, 91
- pipeline bubble, 91
- pipeline hazard, 91
- PIT, 116
- pitch, 5
- pixel, 78
- pixels, 76
- plant, 55
- pointer, 103
- pointers, 101
- popping off a stack, 102
- port, 8
- portable, 116
- power, 123
- power amplifier, 107
- PowerPC, 75, 101
- powerset, 153
- pre, 67
- precedence constraints, 137
- preconditions, 137
- predicate, 27
- preemption, 137
- preemptive scheduler, 137
- printed circuit board, 99
- printer port, 113
- probabilities, 35
- process, 133
- processor assignment, 137
- processor realization, 74
- program counter, 89
- programmable DSP, 76
- programmable DSP processors, 77
- programmable interval timer, 116
- projection, 156
- proportional control, 14
- proportional-integrator (PI) controller, 20
- PThreads, 133
- pull-up resistor, 109, 110
- pure signal, 22
- pushing onto a stack, 102
- quantization noise, 121
- quantized, 121
- Rabbit 2000, 75
- Rabbit Semiconductor, 75
- radar, 77
- RAM, 96
- random access memory, 96
- range, 155
- reaction, 23
- read-only memory, 97
- real numbers, 153
- real-time programs, 137
- receptive, 33
- recursive programs, 102
- reduced instruction set computers, 84

- redundant, 108
- refinement, 46
- refresh of DRAM, 96
- register bank, 89
- register file, 96
- regularizing Zeno systems, 66
- relation, 154
- release time, 137
- reserved, 98
- resolution, 78
- response time, 137
- restriction, 156
- RGB, 85
- RGBA, 79
- RISC, 84
- roll, 4
- ROM, 97
- rounding, 89
- RS-232, 111
- RS-422, 112
- RS-423, 112
- run time, 137
- run-time scheduler, 137

- S28211, 76
- safety-critical system, 117
- safety-critical systems, 137
- sample period, 121
- sample rate, 75, 78
- sampling theorem, 126
- Samsung, 75
- scheduler, 137
- scheduling, 137
- scheduling decision, 137
- Schmitt trigger, 108
- Schmitt, Otto H., 108
- scientific instrumentation, 77
- scratchpad, 99
- SCSI, 113
- segmentation fault, 103
- self transition, 27
- semaphore, 133
- sensor, 55
- sensor network, 74
- sequence, 157
- serial interface, 110
- serial peripheral interface bus, 113
- set, 153
- set action, 46
- set of all functions, 155
- set subtraction, 153
- setpoint, 28
- shading, 81
- shared memory, 133
- shared variable, 116
- Sharp, 75
- short circuit, 108
- signal, 75
- signal conditioning, 122
- signal processing, 75
- signal to noise ratio, 123
- Simulink, 4
- single-board computer, 106
- singleton set, 154
- six degrees of freedom, 4
- sleep mode, 74
- slow motion video, 79
- SNR, 123
- soft core, 93
- software interrupt, 114
- software-defined radio, 76
- sonar, 77
- SparcTM processors, 86
- specification, 36
- speculative execution, 92
- speech synthesis, 76
- SPI, 113
- spontaneous, 33, 34
- squid nerves, 108
- SR, 67
- SRAM, 96
- ST Microelectronics, 75
- stable system, 12
- stack, 102
- stack overflow, 102

- stack pointer, 102
- Stanford University, 75
- start bit, 112
- start time, 137
- state, 25
- state machine, 26
- state refinement, 46
- state space, 26
- Statecharts, 31
- STATEMATE, 31
- static assignment scheduler, 137
- static order scheduler, 137
- static properties, 3
- static RAM, 96
- static variable, 104
- Stellaris, 106, 107
- Stellaris®, 99
- step, 32
- stochastic model, 35
- stop bit, 112
- stop the world, 103
- stuttering, 33, 34
- subset, 153
- subword parallelism, 86
- Sun Microsystems, 86
- SuperH, 75
- superposition, 11
- superscalar, 86
- supervisory control, 55, 56
- supply voltage, 106
- surjective, 155
- surveillance, 74, 77, 87
- symmetric FIR filter, 85
- SyncCharts, 31
- synchronous-reactive, 67
- system, 3
- system clock, 133
- system identification, 77
- SysTick, 115, 130
- tap values, 77
- tapped delay line, 77, 78
- telephony, 75
- teletype, 111
- television, 87
- Texas Instruments, 75, 76, 84, 85, 87, 92
- thermostat, 28, 45
- thread, 133
- thread library, 81
- time-invariant systems, 12
- timed automata, 48
- timer, 116
- timing constraints, 137
- TMS320, 84, 85, 87
- TMS32010, 76
- torque, 6
- Toshiba, 97
- total function, 154
- trace, 37
- tracking error, 15
- transfer function, 124
- transformer, 108
- transistor, 109
- transition, 26, 45
- transition function, 32
- transition relation, 35
- TriMedia, 87
- tristate, 110
- truncation, 89
- tuple, 154
- twos-complement, 88
- UART, 111
- uint8_t, 111
- ultrasound, 75
- UML, 31
- undesired part, 122
- unified modeling language, 31
- unit delay, 78
- unit step, 12
- universal asynchronous receiver/transmitter, 111
- universal serial bus, 112
- unsigned integer, 111
- uPD7720, 76
- update function, 32
- update relation, 35

USB, [99](#), [112](#)

very large instruction word, [82](#), [86](#)

video, [76](#)

video analytics, [77](#)

virtual memory, [98](#)

Viterbi decoding, [84](#)

VLIW, [86](#)

VLSI Technology, [75](#)

voice over IP, [87](#)

voiceband data modems, [76](#)

volatile, [95](#)

volatile keyword, [115](#)

volatile memory, [96](#)

von Neumann, [158](#)

von Neumann numbers, [158](#)

Western Electric, [76](#)

wired AND, [109](#)

wired NAND, [109](#)

wired OR, [109](#)

Wollan, Vegard, [75](#)

word, [96](#)

writeback pipeline stage, [90](#)

x86, [73](#), [74](#), [76](#), [101](#)

x86-64, [76](#)

Yamaha, [75](#)

yaw, [5](#)

Z80, [75](#)

Zeno, [66](#)

zero-overhead loop, [84](#)

Zilog Z80, [75](#)